The Art of Assembly Language
(Brief Contents)

Forward Why Would Anyone Learn This Stuff? 1
Section One: 9
Chapter One Data Representation 11
Chapter Two Boolean Algebra 43
Chapter Three System Organization 83
Chapter Four Memory Layout and Access 145
Section Two: 193
Chapter Five Variables and Data Structures 195
Chapter Six The 80x86 Instruction Set 243
Chapter Seven The UCR Standard Library 333
Chapter Eight MASM: Directives & Pseudo-Opcodes 355
Chapter Nine Arithmetic and Logical Operations 459
Chapter 10 Control Structures 521
Chapter 11 Procedures and Functions 565
Section Three: 637
Chapter 12 Procedures: Advanced Topics 639
Chapter 13 MS-DOS, PC-BIOS, and File I/O 699
Chapter 14 Floating Point Arithmetic 771
Chapter 15 Strings and Character Sets 819
Chapter 16 Pattern Matching 883
Section Four: 993
Chapter 17 Interrupts, Traps, and Exceptions 995
Chapter 18 Resident Programs 1025
Chapter 19 Processes, Coroutines, and Concurrency 1065
Section Five: 1151
Chapter 20 The PC Keyboard 1153
Chapter 21 The PC Parallel Ports 1199
Chapter 22 The PC Serial Ports 1223
Chapter 23 The PC Video Display 1247
Chapter 24 The PC Game Adapter 1255
Section Six: 1309
Chapter 25 Optimizing Your Programs 1311
Section Seven: 1343
Appendix A: ASCII/IBM Character Set 1345
Appendix B: Annotated Bibliography 1347

This document was created with FrameMaker 4.0.2

Appendix C: Keyboard Scan Codes 1351
Appendix D: Instruction Set Reference 1361

The Art of Assembly Language
(Full Contents)

Forward Why Would Anyone Learn This Stuff? 1
1 What's Wrong With Assembly Language 1
2 What's Right With Assembly Language? 4
3 Organization of This Text and Pedagogical Concerns 5
4 Obtaining Program Source Listings and Other Materials in This Text 7
Section One: 9
Machine Organization 9
Chapter One Data Representation 11
1.0 Chapter Overview 11
1.1 Numbering Systems 11
1.1.1 A Review of the Decimal System 11
1.1.2 The Binary Numbering System 12
1.1.3 Binary Formats 13
1.2 Data Organization 13
1.2.1 Bits 14
1.2.2 Nibbles 14
1.2.3 Bytes 14
1.2.4 Words 15
1.2.5 Double Words 16
1.3 The Hexadecimal Numbering System 17
1.4 Arithmetic Operations on Binary and Hexadecimal Numbers 19
1.5 Logical Operations on Bits 20
1.6 Logical Operations on Binary Numbers and Bit Strings 22
1.7 Signed and Unsigned Numbers 23
1.8 Sign and Zero Extension 25
1.9 Shifts and Rotates 26
1.10 Bit Fields and Packed Data 28
1.11 The ASCII Character Set 28
1.12 Summary 31
1.13 Laboratory Exercises 33
1.13.1 Installing the Software 33
1.13.2 Data Conversion Exercises 34
1.13.3 Logical Operations Exercises 35
1.13.4 Sign and Zero Extension Exercises 36
1.13.5 Packed Data Exercises 37
1.14 Questions 38
1.15 Programming Projects 41
Chapter Two Boolean Algebra 43
2.0 Chapter Overview 43
2.1 Boolean Algebra 43

This document was created with FrameMaker 4.0.2

2.2 Boolean Functions and Truth Tables 45
2.3 Algebraic Manipulation of Boolean Expressions 48
2.4 Canonical Forms 49
2.5 Simplification of Boolean Functions 52
2.6 What Does This Have To Do With Computers, Anyway? 59
2.6.1 Correspondence Between Electronic Circuits and Boolean Functions 59
2.6.2 Combinatorial Circuits 60
2.6.3 Sequential and Clocked Logic 62
2.7 Okay, What Does It Have To Do With Programming, Then? 64
2.8 Generic Boolean Functions 65
2.9 Laboratory Exercises 69
2.9.1 Truth Tables and Logic Equations Exercises 70
2.9.2 Canonical Logic Equations Exercises 71
2.9.3 Optimization Exercises 72
2.9.4 Logic Evaluation Exercises 72
2.10 Programming Projects 77
2.11 Summary 78
2.12 Questions 80
Chapter Three System Organization 83
3.0 Chapter Overview 83
3.1 The Basic System Components 83
3.1.1 The System Bus 84
3.1.1.1 The Data Bus 84
3.1.1.2 The Address Bus 86
3.1.1.3 The Control Bus 86
3.1.2 The Memory Subsystem 87
3.1.3 The I/O Subsystem 92
3.2 System Timing 92
3.2.1 The System Clock 92
3.2.2 Memory Access and the System Clock 93
3.2.3 Wait States 95
3.2.4 Cache Memory 96
3.3 The 886, 8286, 8486, and 8686 “Hypothetical” Processors 99
3.3.1 CPU Registers 99
3.3.2 The Arithmetic & Logical Unit 100
3.3.3 The Bus Interface Unit 100
3.3.4 The Control Unit and Instruction Sets 100
3.3.5 The x86 Instruction Set 102
3.3.6 Addressing Modes on the x86 103
3.3.7 Encoding x86 Instructions 104
3.3.8 Step-by-Step Instruction Execution 107
3.3.9 The Differences Between the x86 Processors 109
3.3.10 The 886 Processor 110
3.3.11 The 8286 Processor 110
3.3.12 The 8486 Processor 116
3.3.12.1 The 8486 Pipeline 117
3.3.12.2 Stalls in a Pipeline 118
3.3.12.3 Cache, the Prefetch Queue, and the 8486 119

3.3.12.4 Hazards on the 8486 122
3.3.13 The 8686 Processor 123
3.4 I/O (Input/Output) 124
3.5 Interrupts and Polled I/O 126
3.6 Laboratory Exercises 128
3.6.1 The SIMx86 Program – Some Simple x86 Programs 128
3.6.2 Simple I/O-Mapped Input/Output Operations 131
3.6.3 Memory Mapped I/O 132
3.6.4 DMA Exercises 133
3.6.5 Interrupt Driven I/O Exercises 134
3.6.6 Machine Language Programming & Instruction Encoding Exercises 135
3.6.7 Self Modifying Code Exercises 136
3.7 Programming Projects 138
3.8 Summary 139
3.9 Questions 142
Chapter Four Memory Layout and Access 145
4.0 Chapter Overview 145
4.1 The 80x86 CPUs:A Programmer's View 145
4.1.1 8086 General Purpose Registers 146
4.1.2 8086 Segment Registers 147
4.1.3 8086 Special Purpose Registers 148
4.1.4 80286 Registers 148
4.1.5 80386/80486 Registers 149
4.2 80x86 Physical Memory Organization 150
4.3 Segments on the 80x86 151
4.4 Normalized Addresses on the 80x86 154
4.5 Segment Registers on the 80x86 155
4.6 The 80x86 Addressing Modes 155
4.6.1 8086 Register Addressing Modes 156
4.6.2 8086 Memory Addressing Modes 156
4.6.2.1 The Displacement Only Addressing Mode 156
4.6.2.2 The Register Indirect Addressing Modes 158
4.6.2.3 Indexed Addressing Modes 159
4.6.2.4 Based Indexed Addressing Modes 160
4.6.2.5 Based Indexed Plus Displacement Addressing Mode 160
4.6.2.6 An Easy Way to Remember the 8086 Memory Addressing Modes 162
4.6.2.7 Some Final Comments About 8086 Addressing Modes 162
4.6.3 80386 Register Addressing Modes 163
4.6.4 80386 Memory Addressing Modes 163
4.6.4.1 Register Indirect Addressing Modes 163
4.6.4.2 80386 Indexed, Base/Indexed, and Base/Indexed/Disp Addressing Modes 164
4.6.4.3 80386 Scaled Indexed Addressing Modes 165
4.6.4.4 Some Final Notes About the 80386 Memory Addressing Modes 165
4.7 The 80x86 MOV Instruction 166
4.8 Some Final Comments on the MOV Instructions 169
4.9 Laboratory Exercises 169
4.9.1 The UCR Standard Library for 80x86 Assembly Language Programmers 169
4.9.2 Editing Your Source Files 170

4.9.3 The SHELL.ASM File 170
4.9.4 Assembling Your Code with MASM 172
4.9.5 Debuggers and CodeView™ 173
4.9.5.1 A Quick Look at CodeView 173
4.9.5.2 The Source Window 174
4.9.5.3 The Memory Window 175
4.9.5.4 The Register Window 176
4.9.5.5 The Command Window 176
4.9.5.6 The Output Menu Item 177
4.9.5.7 The CodeView Command Window 177
4.9.5.7.1 The Radix Command (N) 177
4.9.5.7.2 The Assemble Command 178
4.9.5.7.3 The Compare Memory Command 178
4.9.5.7.4 The Dump Memory Command 180
4.9.5.7.5 The Enter Command 181
4.9.5.7.6 The Fill Memory Command 182
4.9.5.7.7 The Move Memory Command 182
4.9.5.7.8 The Input Command 183
4.9.5.7.9 The Output Command 183
4.9.5.7.10 The Quit Command 183
4.9.5.7.11 The Register Command 183
4.9.5.7.12 The Unassemble Command 184
4.9.5.8 CodeView Function Keys 184
4.9.5.9 Some Comments on CodeView Addresses 185
4.9.5.10 A Wrap on CodeView 186
4.9.6 Laboratory Tasks 186
4.10 Programming Projects 187
4.11 Summary 188
4.12 Questions 190
Section Two: 193
Basic Assembly Language 193
Chapter Five Variables and Data Structures 195
5.0 Chapter Overview 195
5.1 Some Additional Instructions: LEA, LES, ADD, and MUL 195
5.2 Declaring Variables in an Assembly Language Program 196
5.3 Declaring and Accessing Scalar Variables 197
5.3.1 Declaring and using BYTE Variables 198
5.3.2 Declaring and using WORD Variables 200
5.3.3 Declaring and using DWORD Variables 201
5.3.4 Declaring and using FWORD, QWORD, and TBYTE Variables 202
5.3.5 Declaring Floating Point Variables with REAL4, REAL8, and REAL10 202
5.4 Creating Your Own Type Names with TYPEDEF 203
5.5 Pointer Data Types 203
5.6 Composite Data Types 206
5.6.1 Arrays 206
5.6.1.1 Declaring Arrays in Your Data Segment 207
5.6.1.2 Accessing Elements of a Single Dimension Array 209
5.6.2 Multidimensional Arrays 210
5.6.2.1 Row Major Ordering 211

5.6.2.2 Column Major Ordering 215
5.6.2.3 Allocating Storage for Multidimensional Arrays 216
5.6.2.4 Accessing Multidimensional Array Elements in Assembly Language 217
5.6.3 Structures 218
5.6.4 Arrays of Structures and Arrays/Structures as Structure Fields 220
5.6.5 Pointers to Structures 221
5.7 Sample Programs 222
5.7.1 Simple Variable Declarations 222
5.7.2 Using Pointer Variables 224
5.7.3 Single Dimension Array Access 226
5.7.4 Multidimensional Array Access 227
5.7.5 Simple Structure Access 229
5.7.6 Arrays of Structures 231
5.7.7 Structures and Arrays as Fields of Another Structure 233
5.7.8 Pointers to Structures and Arrays of Structures 235
5.8 Laboratory Exercises 237
5.9 Programming Projects 238
5.10 Summary 239
5.11 Questions 241
Chapter Six The 80x86 Instruction Set 243
6.0 Chapter Overview 243
6.1 The Processor Status Register (Flags) 244
6.2 Instruction Encodings 245
6.3 Data Movement Instructions 246
6.3.1 The MOV Instruction 246
6.3.2 The XCHG Instruction 247
6.3.3 The LDS, LES, LFS, LGS, and LSS Instructions 248
6.3.4 The LEA Instruction 248
6.3.5 The PUSH and POP Instructions 249
6.3.6 The LAHF and SAHF Instructions 252
6.4 Conversions 252
6.4.1 The MOVZX, MOVSX, CBW, CWD, CWDE, and CDQ Instructions 252
6.4.2 The BSWAP Instruction 254
6.4.3 The XLAT Instruction 255
6.5 Arithmetic Instructions 255
6.5.1 The Addition Instructions: ADD, ADC, INC, XADD, AAA, and DAA 256
6.5.1.1 The ADD and ADC Instructions 256
6.5.1.2 The INC Instruction 258
6.5.1.3 The XADD Instruction 258
6.5.1.4 The AAA and DAA Instructions 258
6.5.2 The Subtraction Instructions: SUB, SBB, DEC, AAS, and DAS 259
6.5.3 The CMP Instruction 261
6.5.4 The CMPXCHG, and CMPXCHG8B Instructions 263
6.5.5 The NEG Instruction 263
6.5.6 The Multiplication Instructions: MUL, IMUL, and AAM 264
6.5.7 The Division Instructions: DIV, IDIV, and AAD 267
6.6 Logical, Shift, Rotate and Bit Instructions 269
6.6.1 The Logical Instructions: AND, OR, XOR, and NOT 269
6.6.2 The Shift Instructions: SHL/SAL, SHR, SAR, SHLD, and SHRD 270

6.6.2.1 SHL/SAL 271
6.6.2.2 SAR 272
6.6.2.3 SHR 273
6.6.2.4 The SHLD and SHRD Instructions 274
6.6.3 The Rotate Instructions: RCL, RCR, ROL, and ROR 276
6.6.3.1 RCL 277
6.6.3.2 RCR 277
6.6.3.3 ROL 278
6.6.3.4 ROR 278
6.6.4 The Bit Operations 279
6.6.4.1 TEST 280
6.6.4.2 The Bit Test Instructions: BT, BTS, BTR, and BTC 280
6.6.4.3 Bit Scanning: BSF and BSR 281
6.6.5 The "Set on Condition" Instructions 281
6.7 I/O Instructions 284
6.8 String Instructions 284
6.9 Program Flow Control Instructions 286
6.9.1 Unconditional Jumps 286
6.9.2 The CALL and RET Instructions 289
6.9.3 The INT, INTO, BOUND, and IRET Instructions 292
6.9.4 The Conditional Jump Instructions 296
6.9.5 The JCXZ/JECXZ Instructions 299
6.9.6 The LOOP Instruction 300
6.9.7 The LOOPE/LOOPZ Instruction 300
6.9.8 The LOOPNE/LOOPNZ Instruction 301
6.10 Miscellaneous Instructions 302
6.11 Sample Programs 303
6.11.1 Simple Arithmetic I 303
6.11.2 Simple Arithmetic II 305
6.11.3 Logical Operations 306
6.11.4 Shift and Rotate Operations 308
6.11.5 Bit Operations and SETcc Instructions 310
6.11.6 String Operations 312
6.11.7 Conditional Jumps 313
6.11.8 CALL and INT Instructions 315
6.11.9 Conditional Jumps I 317
6.11.10 Conditional Jump Instructions II 318
6.12 Laboratory Exercises 320
6.12.1 The IBM/L System 320
6.12.2 IBM/L Exercises 327
6.13 Programming Projects 327
6.14 Summary 328
6.15 Questions 331
Chapter Seven The UCR Standard Library 333
7.0 Chapter Overview 333
7.1 An Introduction to the UCR Standard Library 333
7.1.1 Memory Management Routines: MEMINIT, MALLOC, and FREE 334
7.1.2 The Standard Input Routines: GETC, GETS, GETSM 334
7.1.3 The Standard Output Routines: PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT, and PRINTF 336

7.1.4 Formatted Output Routines: Putisize, Putusize, Putlsize, and Putulsize 340
7.1.5 Output Field Size Routines: Isize, Usize, and Lsize 340
7.1.6 Conversion Routines: ATOx, and xTOA 341
7.1.7 Routines that Test Characters for Set Membership 342
7.1.8 Character Conversion Routines: ToUpper, ToLower 343
7.1.9 Random Number Generation: Random, Randomize 343
7.1.10 Constants, Macros, and other Miscellany 344
7.1.11 Plus more! 344
7.2 Sample Programs 344
7.2.1 Stripped SHELL.ASM File 345
7.2.2 Numeric I/O 345
7.3 Laboratory Exercises 348
7.3.1 Obtaining the UCR Standard Library 348
7.3.2 Unpacking the Standard Library 349
7.3.3 Using the Standard Library 349
7.3.4 The Standard Library Documentation Files 350
7.4 Programming Projects 351
7.5 Summary 351
7.6 Questions 353
Chapter Eight MASM: Directives & Pseudo-Opcodes 355
8.0 Chapter Overview 355
8.1 Assembly Language Statements 355
8.2 The Location Counter 357
8.3 Symbols 358
8.4 Literal Constants 359
8.4.1 Integer Constants 360
8.4.2 String Constants 361
8.4.3 Real Constants 361
8.4.4 Text Constants 362
8.5 Declaring Manifest Constants Using Equates 362
8.6 Processor Directives 364
8.7 Procedures 365
8.8 Segments 366
8.8.1 Segment Names 367
8.8.2 Segment Loading Order 368
8.8.3 Segment Operands 369
8.8.3.1 The ALIGN Type 369
8.8.3.2 The COMBINE Type 373
8.8.4 The CLASS Type 374
8.8.5 The Read-only Operand 375
8.8.6 The USE16, USE32, and FLAT Options 375
8.8.7 Typical Segment Definitions 376
8.8.8 Why You Would Want to Control the Loading Order 376
8.8.9 Segment Prefixes 377
8.8.10 Controlling Segments with the ASSUME Directive 377
8.8.11 Combining Segments: The GROUP Directive 380
8.8.12 Why Even Bother With Segments? 383
8.9 The END Directive 384

8.10 Variables 384
8.11 Label Types 385
8.11.1 How to Give a Symbol a Particular Type 385
8.11.2 Label Values 386
8.11.3 Type Conflicts 386
8.12 Address Expressions 387
8.12.1 Symbol Types and Addressing Modes 387
8.12.2 Arithmetic and Logical Operators 388
8.12.3 Coercion 390
8.12.4 Type Operators 392
8.12.5 Operator Precedence 396
8.13 Conditional Assembly 397
8.13.1 IF Directive 398
8.13.2 IFE directive 399
8.13.3 IFDEF and IFNDEF 399
8.13.4 IFB, IFNB 399
8.13.5 IFIDN, IFDIF, IFIDNI, and IFDIFI 400
8.14 Macros 400
8.14.1 Procedural Macros 400
8.14.2 Macros vs. 80x86 Procedures 404
8.14.3 The LOCAL Directive 406
8.14.4 The EXITM Directive 406
8.14.5 Macro Parameter Expansion and Macro Operators 407
8.14.6 A Sample Macro to Implement For Loops 409
8.14.7 Macro Functions 413
8.14.8 Predefined Macros, Macro Functions, and Symbols 414
8.14.9 Macros vs. Text Equates 418
8.14.10 Macros: Good and Bad News 419
8.15 Repeat Operations 420
8.16 The FOR and FORC Macro Operations 421
8.17 The WHILE Macro Operation 422
8.18 Macro Parameters 422
8.19 Controlling the Listing 424
8.19.1 The ECHO and %OUT Directives 424
8.19.2 The TITLE Directive 424
8.19.3 The SUBTTL Directive 424
8.19.4 The PAGE Directive 424
8.19.5 The .LIST, .NOLIST, and .XLIST Directives 425
8.19.6 Other Listing Directives 425
8.20 Managing Large Programs 425
8.20.1 The INCLUDE Directive 426
8.20.2 The PUBLIC, EXTERN, and EXTRN Directives 427
8.20.3 The EXTERNDEF Directive 428
8.21 Make Files 429
8.22 Sample Program 432
8.22.1 EX8.MAK 432
8.22.2 Matrix.A 432
8.22.3 EX8.ASM 433
8.22.4 GETI.ASM 442

8.22.5 GetArray.ASM 443
8.22.6 XProduct.ASM 445
8.23 Laboratory Exercises 447
8.23.1 Near vs. Far Procedures 447
8.23.2 Data Alignment Exercises 448
8.23.3 Equate Exercise 449
8.23.4 IFDEF Exercise 450
8.23.5 Make File Exercise 451
8.24 Programming Projects 453
8.25 Summary 453
8.26 Questions 456
Chapter Nine Arithmetic and Logical Operations 459
9.0 Chapter Overview 459
9.1 Arithmetic Expressions 460
9.1.1 Simple Assignments 460
9.1.2 Simple Expressions 460
9.1.3 Complex Expressions 462
9.1.4 Commutative Operators 466
9.2 Logical (Boolean) Expressions 467
9.3 Multiprecision Operations 470
9.3.1 Multiprecision Addition Operations 470
9.3.2 Multiprecision Subtraction Operations 472
9.3.3 Extended Precision Comparisons 473
9.3.4 Extended Precision Multiplication 475
9.3.5 Extended Precision Division 477
9.3.6 Extended Precision NEG Operations 480
9.3.7 Extended Precision AND Operations 481
9.3.8 Extended Precision OR Operations 482
9.3.9 Extended Precision XOR Operations 482
9.3.10 Extended Precision NOT Operations 482
9.3.11 Extended Precision Shift Operations 482
9.3.12 Extended Precision Rotate Operations 484
9.4 Operating on Different Sized Operands 485
9.5 Machine and Arithmetic Idioms 486
9.5.1 Multiplying Without MUL and IMUL 487
9.5.2 Division Without DIV and IDIV 488
9.5.3 Using AND to Compute Remainders 488
9.5.4 Implementing Modulo-n Counters with AND 489
9.5.5 Testing an Extended Precision Value for 0FFFF..FFh 489
9.5.6 TEST Operations 489
9.5.7 Testing Signs with the XOR Instruction 490
9.6 Masking Operations 490
9.6.1 Masking Operations with the AND Instruction 490
9.6.2 Masking Operations with the OR Instruction 491
9.7 Packing and Unpacking Data Types 491
9.8 Tables 493
9.8.1 Function Computation via Table Look Up 493
9.8.2 Domain Conditioning 496

9.8.3 Generating Tables 497
9.9 Sample Programs 498
9.9.1 Converting Arithmetic Expressions to Assembly Language 498
9.9.2 Boolean Operations Example 500
9.9.3 64-bit Integer I/O 503
9.9.4 Packing and Unpacking Date Data Types 506
9.10 Laboratory Exercises 509
9.10.1 Debugging Programs with CodeView 509
9.10.2 Debugging Strategies 511
9.10.2.1 Locating Infinite Loops 511
9.10.2.2 Incorrect Computations 512
9.10.2.3 Illegal Instructions/Infinite Loops Part II 513
9.10.3 Debug Exercise I: Using CodeView to Find Bugs in a Calculation 513
9.10.4 Software Delay Loop Exercises 515
9.11 Programming Projects 516
9.12 Summary 516
9.13 Questions 518
Chapter 10 Control Structures 521
10.0 Chapter Overview 521
10.1 Introduction to Decisions 521
10.2 IF..THEN..ELSE Sequences 522
10.3 CASE Statements 525
10.4 State Machines and Indirect Jumps 529
10.5 Spaghetti Code 531
10.6 Loops 531
10.6.1 While Loops 532
10.6.2 Repeat..Until Loops 532
10.6.3 LOOP..ENDLOOP Loops 533
10.6.4 FOR Loops 533
10.7 Register Usage and Loops 534
10.8 Performance Improvements 535
10.8.1 Moving the Termination Condition to the End of a Loop 535
10.8.2 Executing the Loop Backwards 537
10.8.3 Loop Invariant Computations 538
10.8.4 Unraveling Loops 539
10.8.5 Induction Variables 540
10.8.6 Other Performance Improvements 541
10.9 Nested Statements 542
10.10 Timing Delay Loops 544
10.11 Sample Program 547
10.12 Laboratory Exercises 552
10.12.1 The Physics of Sound 552
10.12.2 The Fundamentals of Music 553
10.12.3 The Physics of Music 554
10.12.4 The 8253/8254 Timer Chip 555
10.12.5 Programming the Timer Chip to Produce Musical Tones 555
10.12.6 Putting it All Together 556

10.12.7 Amazing Grace Exercise 557
10.13 Programming Projects 558
10.14 Summary 559
10.15 Questions 561
Chapter 11 Procedures and Functions 565
11.0 Chapter Overview 565
11.1 Procedures 566
11.2 Near and Far Procedures 568
11.2.1 Forcing NEAR or FAR CALLs and Returns 568
11.2.2 Nested Procedures 569
11.3 Functions 572
11.4 Saving the State of the Machine 572
11.5 Parameters 574
11.5.1 Pass by Value 574
11.5.2 Pass by Reference 575
11.5.3 Pass by Value-Returned 575
11.5.4 Pass by Result 576
11.5.5 Pass by Name 576
11.5.6 Pass by Lazy-Evaluation 577
11.5.7 Passing Parameters in Registers 578
11.5.8 Passing Parameters in Global Variables 580
11.5.9 Passing Parameters on the Stack 581
11.5.10 Passing Parameters in the Code Stream 590
11.5.11 Passing Parameters via a Parameter Block 598
11.6 Function Results 600
11.6.1 Returning Function Results in a Register 601
11.6.2 Returning Function Results on the Stack 601
11.6.3 Returning Function Results in Memory Locations 602
11.7 Side Effects 602
11.8 Local Variable Storage 604
11.9 Recursion 606
11.10 Sample Program 610
11.11 Laboratory Exercises 618
11.11.1 Ex11_1.cpp 619
11.11.2 Ex11_1.asm 621
11.11.3 EX11_1a.asm 625
11.12 Programming Projects 632
11.13 Summary 633
11.14 Questions 635
Section Three: 637
Intermediate Level Assembly Language Programming 637
Chapter 12 Procedures: Advanced Topics 639
12.0 Chapter Overview 639
12.1 Lexical Nesting, Static Links, and Displays 639
12.1.1 Scope 640

12.1.2 Unit Activation, Address Binding, and Variable Lifetime 642
12.1.3 Static Links 642
12.1.4 Accessing Non-Local Variables Using Static Links 647
12.1.5 The Display 648
12.1.6 The 80286 ENTER and LEAVE Instructions 650
12.2 Passing Variables at Different Lex Levels as Parameters. 652
12.2.1 Passing Parameters by Value in a Block Structured Language 652
12.2.2 Passing Parameters by Reference, Result, and Value-Result in a Block Structured Language 653
12.2.3 Passing Parameters by Name and Lazy-Evaluation in a Block Structured Language 654
12.3 Passing Parameters as Parameters to Another Procedure 655
12.3.1 Passing Reference Parameters to Other Procedures 656
12.3.2 Passing Value-Result and Result Parameters as Parameters 657
12.3.3 Passing Name Parameters to Other Procedures 657
12.3.4 Passing Lazy Evaluation Parameters as Parameters 658
12.3.5 Parameter Passing Summary 658
12.4 Passing Procedures as Parameters 659
12.5 Iterators 663
12.5.1 Implementing Iterators Using In-Line Expansion 664
12.5.2 Implementing Iterators with Resume Frames 666
12.6 Sample Programs 669
12.6.1 An Example of an Iterator 669
12.6.2 Another Iterator Example 673
12.7 Laboratory Exercises 678
12.7.1 Iterator Exercise 678
12.7.2 The 80x86 Enter and Leave Instructions 684
12.7.3 Parameter Passing Exercises 690
12.8 Programming Projects 695
12.9 Summary 697
12.10 Questions 698
Chapter 13 MS-DOS, PC-BIOS, and File I/O 699
13.0 Chapter Overview 700
13.1 The IBM PC BIOS 701
13.2 An Introduction to the BIOS' Services 701
13.2.1 INT 5- Print Screen 702
13.2.2 INT 10h - Video Services 702
13.2.3 INT 11h - Equipment Installed 704
13.2.4 INT 12h - Memory Available 704
13.2.5 INT 13h - Low Level Disk Services 704
13.2.6 INT 14h - Serial I/O 706
13.2.6.1 AH=0: Serial Port Initialization 706
13.2.6.2 AH=1: Transmit a Character to the Serial Port 707
13.2.6.3 AH=2: Receive a Character from the Serial Port 707
13.2.6.4 AH=3: Serial Port Status 707
13.2.7 INT 15h - Miscellaneous Services 708
13.2.8 INT 16h - Keyboard Services 708
13.2.8.1 AH=0: Read a Key From the Keyboard 709
13.2.8.2 AH=1: See if a Key is Available at the Keyboard 709
13.2.8.3 AH=2: Return Keyboard Shift Key Status 710
13.2.9 INT 17h - Printer Services 710

13.2.9.1 AH=0: Print a Character 711
13.2.9.2 AH=1: Initialize Printer 711
13.2.9.3 AH=2: Return Printer Status 711
13.2.10 INT 18h - Run BASIC 712
13.2.11 INT 19h - Reboot Computer 712
13.2.12 INT 1Ah - Real Time Clock 712
13.2.12.1 AH=0: Read the Real Time Clock 712
13.2.12.2 AH=1: Setting the Real Time Clock 713
13.3 An Introduction to MS-DOS™ 713
13.3.1 MS-DOS Calling Sequence 714
13.3.2 MS-DOS Character Oriented Functions 714
13.3.3 MS-DOS Drive Commands 716
13.3.4 MS-DOS “Obsolete” Filing Calls 717
13.3.5 MS-DOS Date and Time Functions 718
13.3.6 MS-DOS Memory Management Functions 718
13.3.6.1 Allocate Memory 719
13.3.6.2 Deallocate Memory 719
13.3.6.3 Modify Memory Allocation 719
13.3.6.4 Advanced Memory Management Functions 720
13.3.7 MS-DOS Process Control Functions 721
13.3.7.1 Terminate Program Execution 721
13.3.7.2 Terminate, but Stay Resident 721
13.3.7.3 Execute a Program 722
13.3.8 MS-DOS “New” Filing Calls 725
13.3.8.1 Open File 725
13.3.8.2 Create File 726
13.3.8.3 Close File 727
13.3.8.4 Read From a File 727
13.3.8.5 Write to a File 728
13.3.8.6 Seek (Move File Pointer) 728
13.3.8.7 Set Disk Transfer Address (DTA) 729
13.3.8.8 Find First File 729
13.3.8.9 Find Next File 730
13.3.8.10 Delete File 730
13.3.8.11 Rename File 730
13.3.8.12 Change/Get File Attributes 731
13.3.8.13 Get/Set File Date and Time 731
13.3.8.14 Other DOS Calls 732
13.3.9 File I/O Examples 734
13.3.9.1 Example #1: A Hex Dump Utility 734
13.3.9.2 Example #2: Upper Case Conversion 735
13.3.10 Blocked File I/O 737
13.3.11 The Program Segment Prefix (PSP) 739
13.3.12 Accessing Command Line Parameters 742
13.3.13 ARGC and ARGV 750
13.4 UCR Standard Library File I/O Routines 751
13.4.1 Fopen 751
13.4.2 Fcreate 752
13.4.3 Fclose 752
13.4.4 Fflush 752
13.4.5 Fgetc 752
13.4.6 Fread 753
13.4.7 Fputc 753

13.4.8 Fwrite 753
13.4.9 Redirecting I/O Through the StdLib File I/O Routines 753
13.4.10 A File I/O Example 755
13.5 Sample Program 758
13.6 Laboratory Exercises 763
13.7 Programming Projects 768
13.8 Summary 768
13.9 Questions 770
Chapter 14 Floating Point Arithmetic 771
14.0 Chapter Overview 771
14.1 The Mathematics of Floating Point Arithmetic 771
14.2 IEEE Floating Point Formats 774
14.3 The UCR Standard Library Floating Point Routines 777
14.3.1 Load and Store Routines 778
14.3.2 Integer/Floating Point Conversion 779
14.3.3 Floating Point Arithmetic 780
14.3.4 Float/Text Conversion and Printff 780
14.4 The 80x87 Floating Point Coprocessors 781
14.4.1 FPU Registers 781
14.4.1.1 The FPU Data Registers 782
14.4.1.2 The FPU Control Register 782
14.4.1.3 The FPU Status Register 785
14.4.2 FPU Data Types 788
14.4.3 The FPU Instruction Set 789
14.4.4 FPU Data Movement Instructions 789
14.4.4.1 The FLD Instruction 789
14.4.4.2 The FST and FSTP Instructions 790
14.4.4.3 The FXCH Instruction 790
14.4.5 Conversions 791
14.4.5.1 The FILD Instruction 791
14.4.5.2 The FIST and FISTP Instructions 791
14.4.5.3 The FBLD and FBSTP Instructions 792
14.4.6 Arithmetic Instructions 792
14.4.6.1 The FADD and FADDP Instructions 792
14.4.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions 793
14.4.6.3 The FMUL and FMULP Instructions 794
14.4.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions 794
14.4.6.5 The FSQRT Instruction 795
14.4.6.6 The FSCALE Instruction 795
14.4.6.7 The FPREM and FPREM1 Instructions 795
14.4.6.8 The FRNDINT Instruction 796
14.4.6.9 The FXTRACT Instruction 796
14.4.6.10 The FABS Instruction 796
14.4.6.11 The FCHS Instruction 797
14.4.7 Comparison Instructions 797
14.4.7.1 The FCOM, FCOMP, and FCOMPP Instructions 797
14.4.7.2 The FUCOM, FUCOMP, and FUCOMPP Instructions 798
14.4.7.3 The FTST Instruction 798
14.4.7.4 The FXAM Instruction 798
14.4.8 Constant Instructions 798

14.4.9 Transcendental Instructions 799
14.4.9.1 The F2XM1 Instruction 799
14.4.9.2 The FSIN, FCOS, and FSINCOS Instructions 799
14.4.9.3 The FPTAN Instruction 799
14.4.9.4 The FPATAN Instruction 800
14.4.9.5 The FYL2X and FYL2XP1 Instructions 800
14.4.10 Miscellaneous instructions 800
14.4.10.1 The FINIT and FNINIT Instructions 800
14.4.10.2 The FWAIT Instruction 801
14.4.10.3 The FLDCW and FSTCW Instructions 801
14.4.10.4 The FCLEX and FNCLEX Instructions 801
14.4.10.5 The FLDENV, FSTENV, and FNSTENV Instructions 801
14.4.10.6 The FSAVE, FNSAVE, and FRSTOR Instructions 802
14.4.10.7 The FSTSW and FNSTSW Instructions 803
14.4.10.8 The FINCSTP and FDECSTP Instructions 803
14.4.10.9 The FNOP Instruction 803
14.4.10.10 The FFREE Instruction 803
14.4.11 Integer Operations 803
14.5 Sample Program: Additional Trigonometric Functions 804
14.6 Laboratory Exercises 810
14.6.1 FPU vs StdLib Accuracy 811
14.7 Programming Projects 814
14.8 Summary 814
14.9 Questions 817
Chapter 15 Strings and Character Sets 819
15.0 Chapter Overview 819
15.1 The 80x86 String Instructions 819
15.1.1 How the String Instructions Operate 819
15.1.2 The REP/REPE/REPZ and REPNZ/REPNE Prefixes 820
15.1.3 The Direction Flag 821
15.1.4 The MOVS Instruction 822
15.1.5 The CMPS Instruction 826
15.1.6 The SCAS Instruction 828
15.1.7 The STOS Instruction 828
15.1.8 The LODS Instruction 829
15.1.9 Building Complex String Functions from LODS and STOS 830
15.1.10 Prefixes and the String Instructions 830
15.2 Character Strings 831
15.2.1 Types of Strings 831
15.2.2 String Assignment 832
15.2.3 String Comparison 834
15.3 Character String Functions 835
15.3.1 Substr 835
15.3.2 Index 838
15.3.3 Repeat 840
15.3.4 Insert 841
15.3.5 Delete 843
15.3.6 Concatenation 844
15.4 String Functions in the UCR Standard Library 845

15.4.1 StrBDel, StrBDelm 846
15.4.2 Strcat, Strcatl, Strcatm, Strcatml 847
15.4.3 Strchr 848
15.4.4 Strcmp, Strcmpl, Stricmp, Stricmpl 848
15.4.5 Strcpy, Strcpyl, Strdup, Strdupl 849
15.4.6 Strdel, Strdelm 850
15.4.7 Strins, Strinsl, Strinsm, Strinsml 851
15.4.8 Strlen 852
15.4.9 Strlwr, Strlwrm, Strupr, Struprm 852
15.4.10 Strrev, Strrevm 853
15.4.11 Strset, Strsetm 853
15.4.12 Strspan, Strspanl, Strcspan, Strcspanl 854
15.4.13 Strstr, Strstrl 855
15.4.14 Strtrim, Strtrimm 855
15.4.15 Other String Routines in the UCR Standard Library 856
15.5 The Character Set Routines in the UCR Standard Library 856
15.6 Using the String Instructions on Other Data Types 859
15.6.1 Multi-precision Integer Strings 859
15.6.2 Dealing with Whole Arrays and Records 860
15.7 Sample Programs 860
15.7.1 Find.asm 860
15.7.2 StrDemo.asm 862
15.7.3 Fcmp.asm 865
15.8 Laboratory Exercises 868
15.8.1 MOVS Performance Exercise #1 868
15.8.2 MOVS Performance Exercise #2 870
15.8.3 Memory Performance Exercise 872
15.8.4 The Performance of Length-Prefixed vs. Zero-Terminated Strings 874
15.9 Programming Projects 878
15.10 Summary 878
15.11 Questions 881
Chapter 16 Pattern Matching 883
16.1 An Introduction to Formal Language (Automata) Theory 883
16.1.1 Machines vs. Languages 883
16.1.2 Regular Languages 884
16.1.2.1 Regular Expressions 885
16.1.2.2 Nondeterministic Finite State Automata (NFAs) 887
16.1.2.3 Converting Regular Expressions to NFAs 888
16.1.2.4 Converting an NFA to Assembly Language 890
16.1.2.5 Deterministic Finite State Automata (DFAs) 893
16.1.2.6 Converting a DFA to Assembly Language 895
16.1.3 Context Free Languages 900
16.1.4 Eliminating Left Recursion and Left Factoring CFGs 903
16.1.5 Converting REs to CFGs 905
16.1.6 Converting CFGs to Assembly Language 905
16.1.7 Some Final Comments on CFGs 912
16.1.8 Beyond Context Free Languages 912
16.2 The UCR Standard Library Pattern Matching Routines 913
16.3 The Standard Library Pattern Matching Functions 914

16.3.1 Spancset 914
16.3.2 Brkcset 915
16.3.3 Anycset 915
16.3.4 Notanycset 916
16.3.5 MatchStr 916
16.3.6 MatchiStr 916
16.3.7 MatchToStr 917
16.3.8 MatchChar 917
16.3.9 MatchToChar 918
16.3.10 MatchChars 918
16.3.11 MatchToPat 918
16.3.12 EOS 919
16.3.13 ARB 919
16.3.14 ARBNUM 920
16.3.15 Skip 920
16.3.16 Pos 921
16.3.17 RPos 921
16.3.18 GotoPos 921
16.3.19 RGotoPos 922
16.3.20 SL_Match2 922
16.4 Designing Your Own Pattern Matching Routines 922
16.5 Extracting Substrings from Matched Patterns 925
16.6 Semantic Rules and Actions 929
16.7 Constructing Patterns for the MATCH Routine 933
16.8 Some Sample Pattern Matching Applications 935
16.8.1 Converting Written Numbers to Integers 935
16.8.2 Processing Dates 941
16.8.3 Evaluating Arithmetic Expressions 948
16.8.4 A Tiny Assembler 953
16.8.5 The “MADVENTURE” Game 963
16.9 Laboratory Exercises 979
16.9.1 Checking for Stack Overflow (Infinite Loops) 979
16.9.2 Printing Diagnostic Messages from a Pattern 984
16.10 Programming Projects 988
16.11 Summary 988
16.12 Questions 991
Section Four: 993
Advanced Assembly Language Programming 993
Chapter 17 Interrupts, Traps, and Exceptions 995
17.1 80x86 Interrupt Structure and Interrupt Service Routines (ISRs) 996
17.2 Traps 999
17.3 Exceptions 1000
17.3.1 Divide Error Exception (INT 0) 1000
17.3.2 Single Step (Trace) Exception (INT 1) 1000
17.3.3 Breakpoint Exception (INT 3) 1001
17.3.4 Overflow Exception (INT 4/INTO) 1001
17.3.5 Bounds Exception (INT 5/BOUND) 1001
17.3.6 Invalid Opcode Exception (INT 6) 1004

17.3.7 Coprocessor Not Available (INT 7) 1004
17.4 Hardware Interrupts 1004
17.4.1 The 8259A Programmable Interrupt Controller (PIC) 1005
17.4.2 The Timer Interrupt (INT 8) 1007
17.4.3 The Keyboard Interrupt (INT 9) 1008
17.4.4 The Serial Port Interrupts (INT 0Bh and INT 0Ch) 1008
17.4.5 The Parallel Port Interrupts (INT 0Dh and INT 0Fh) 1008
17.4.6 The Diskette and Hard Drive Interrupts (INT 0Eh and INT 76h) 1009
17.4.7 The Real-Time Clock Interrupt (INT 70h) 1009
17.4.8 The FPU Interrupt (INT 75h) 1009
17.4.9 Nonmaskable Interrupts (INT 2) 1009
17.4.10 Other Interrupts 1009
17.5 Chaining Interrupt Service Routines 1010
17.6 Reentrancy Problems 1012
17.7 The Efficiency of an Interrupt Driven System 1014
17.7.1 Interrupt Driven I/O vs. Polling 1014
17.7.2 Interrupt Service Time 1015
17.7.3 Interrupt Latency 1016
17.7.4 Prioritized Interrupts 1020
17.8 Debugging ISRs 1020
17.9 Summary 1021
Chapter 18 Resident Programs 1025
18.1 DOS Memory Usage and TSRs 1025
18.2 Active vs. Passive TSRs 1029
18.3 Reentrancy 1032
18.3.1 Reentrancy Problems with DOS 1032
18.3.2 Reentrancy Problems with BIOS 1033
18.3.3 Reentrancy Problems with Other Code 1034
18.4 The Multiplex Interrupt (INT 2Fh) 1034
18.5 Installing a TSR 1035
18.6 Removing a TSR 1037
18.7 Other DOS Related Issues 1039
18.8 A Keyboard Monitor TSR 1041
18.9 Semiresident Programs 1055
18.10 Summary 1064
Chapter 19 Processes, Coroutines, and Concurrency 1065
19.1 DOS Processes 1065
19.1.1 Child Processes in DOS 1065
19.1.1.1 Load and Execute 1066
19.1.1.2 Load Program 1068
19.1.1.3 Loading Overlays 1069
19.1.1.4 Terminating a Process 1069
19.1.1.5 Obtaining the Child Process Return Code 1070
19.1.2 Exception Handling in DOS: The Break Handler 1070
19.1.3 Exception Handling in DOS: The Critical Error Handler 1071
19.1.4 Exception Handling in DOS: Traps 1075
19.1.5 Redirection of I/O for Child Processes 1075

19.2 Shared Memory 1078
19.2.1 Static Shared Memory 1078
19.2.2 Dynamic Shared Memory 1088
19.3 Coroutines 1103
19.4 Multitasking 1124
19.4.1 Lightweight and HeavyWeight Processes 1124
19.4.2 The UCR Standard Library Processes Package 1125
19.4.3 Problems with Multitasking 1126
19.4.4 A Sample Program with Threads 1127
19.5 Synchronization 1129
19.5.1 Atomic Operations, Test & Set, and Busy-Waiting 1132
19.5.2 Semaphores 1134
19.5.3 The UCR Standard Library Semaphore Support 1136
19.5.4 Using Semaphores to Protect Critical Regions 1136
19.5.5 Using Semaphores for Barrier Synchronization 1140
19.6 Deadlock 1146
19.7 Summary 1147
Section Five: 1151
The PC's I/O Ports 1151
Chapter 20 The PC Keyboard 1153
20.1 Keyboard Basics 1153
20.2 The Keyboard Hardware Interface 1159
20.3 The Keyboard DOS Interface 1167
20.4 The Keyboard BIOS Interface 1168
20.5 The Keyboard Interrupt Service Routine 1174
20.6 Patching into the INT 9 Interrupt Service Routine 1184
20.7 Simulating Keystrokes 1186
20.7.1 Stuffing Characters in the Type Ahead Buffer 1186
20.7.2 Using the 80x86 Trace Flag to Simulate IN AL, 60H Instructions 1187
20.7.3 Using the 8042 Microcontroller to Simulate Keystrokes 1192
20.8 Summary 1196
Chapter 21 The PC Parallel Ports 1199
21.1 Basic Parallel Port Information 1199
21.2 The Parallel Port Hardware 1201
21.3 Controlling a Printer Through the Parallel Port 1202
21.3.1 Printing via DOS 1203
21.3.2 Printing via BIOS 1203
21.3.3 An INT 17h Interrupt Service Routine 1203
21.4 Inter-Computer Communications on the Parallel Port 1209
21.5 Summary 1222
Chapter 22 The PC Serial Ports 1223
22.1 The 8250 Serial Communications Chip 1223
22.1.1 The Data Register (Transmit/Receive Register) 1224
22.1.2 The Interrupt Enable Register (IER) 1224
22.1.3 The Baud Rate Divisor 1225

22.1.4 The Interrupt Identification Register (IIR) 1226
22.1.5 The Line Control Register 1227
22.1.6 The Modem Control Register 1228
22.1.7 The Line Status Register (LSR) 1229
22.1.8 The Modem Status Register (MSR) 1230
22.1.9 The Auxiliary Input Register 1231
22.2 The UCR Standard Library Serial Communications Support Routines 1231
22.3 Programming the 8250 (Examples from the Standard Library) 1233
22.4 Summary 1244
Chapter 23 The PC Video Display 1247
23.1 Memory Mapped Video 1247
23.2 The Video Attribute Byte 1248
23.3 Programming the Text Display 1249
23.4 Summary 1252
Chapter 24 The PC Game Adapter 1255
24.1 Typical Game Devices 1255
24.2 The Game Adapter Hardware 1257
24.3 Using BIOS' Game I/O Functions 1259
24.4 Writing Your Own Game I/O Routines 1260
24.5 The Standard Game Device Interface (SGDI) 1262
24.5.1 Application Programmer's Interface (API) 1262
24.5.2 Read4Sw 1263
24.5.3 Read4Pots: 1263
24.5.4 ReadPot 1264
24.5.5 Read4: 1264
24.5.6 CalibratePot 1264
24.5.7 TestPotCalibration 1264
24.5.8 ReadRaw 1265
24.5.9 ReadSwitch 1265
24.5.10 Read16Sw 1265
24.5.11 Remove 1265
24.5.12 TestPresence 1265
24.5.13 An SGDI Driver for the Standard Game Adapter Card 1265
24.6 An SGDI Driver for the CH Products' Flight Stick Pro™ 1280
24.7 Patching Existing Games 1293
24.8 Summary 1306
Section Six: 1309
Optimization 1309
Chapter 25 Optimizing Your Programs 1311
25.0 Chapter Overview 1311
25.1 When to Optimize, When Not to Optimize 1311
25.2 How Do You Find the Slow Code in Your Programs? 1313
25.3 Is Optimization Necessary? 1314
25.4 The Three Types of Optimization 1315
25.5 Improving the Implementation of an Algorithm 1317

25.6 Summary 1341
Section Seven: 1343
Appendixes 1343
Appendix A: ASCII/IBM Character Set 1345
Appendix B: Annotated Bibliography 1347
Appendix C: Keyboard Scan Codes 1351
Appendix D: Instruction Set Reference 1361

Why Would Anyone Learn This Stuff? Forward

Amazing! You're actually reading this. That puts you into one of three categories: a student who is being forced to read this stuff for a class, someone who picked up this book by accident (probably because you have yet to be indoctrinated by the world at large), or one of the few who actually have an interest in learning assembly language.

Egads. What kind of book begins this way? What kind of author would begin the book with a forward like this one? Well, the truth is, *I* considered putting this stuff into the first chapter since most people never bother reading the forward. A discussion of what's right and what's wrong with assembly language is very important and sticking it into a chapter might encourage someone to read it. However, I quickly found that university students can skip Chapter One as easily as they can skip a forward, so this stuff wound up in a forward after all.

So why would anyone learn this stuff, anyway? Well, there are several reasons which come to mind:

- Your major requires a course in assembly language; i.e., you're here against your will.
- A programmer where you work quit. Most of the source code left behind was written in assembly language and you were elected to maintain it.
- Your boss has the audacity to insist that you write your code in assembly against your strongest wishes.
- Your programs run just a little too slow, or are a little too large and you think assembly language might help you get your project under control.
- You want to understand how computers actually work.
- You're interested in learning how to write efficient code.
- You want to try something new.

Well, whatever the reason you're here, welcome aboard. Let's take a look at the subject you're about to study.

1 What's Wrong With Assembly Language

Assembly language has a pretty bad reputation. The common impression about assembly language programmers today is that they are all hackers or misguided individuals who need enlightenment. Here are the reasons people give for *not* using assembly[1]:

- Assembly is hard to learn.
- Assembly is hard to read and understand.
- Assembly is hard to debug.
- Assembly is hard to maintain.
- Assembly is hard to write.
- Assembly language programming is time consuming.
- Improved compiler technology has eliminated the need for assembly language.
- Today, machines are so fast that we no longer need to use assembly.
- If you need more speed, you should use a better algorithm rather than switch to assembly language.
- Machines have so much memory today, saving space using assembly is not important.
- Assembly language is not portable.

1. This text will use the terms "Assembly language" and "assembly" interchangeably.

These are some strong statements indeed!

Given that this is a book which teaches assembly language programming, written for college level students, written by someone who appears to know what he's talking about, your natural tendency is to believe something if it appears in print. Having just read the above, you're starting to assume that assembly must be pretty bad. And that, dear friend, is eighty percent of what's wrong with assembly language. That is, people develop some very strong misconceptions about assembly language based on what they've heard from friends, instructors, articles, and books. Oh, assembly language is certainly not perfect. It does have many real faults. Those faults, however, are blown completely out of proportion by those unfamiliar with assembly language. The next time someone starts preaching about the evils of assembly language, ask, "how many years of assembly language programming experience do you have?" Of course assembly is hard to understand *if you don't know it.* It is surprising how many people are willing to speak out against assembly language based only on conversations they've had or articles they've read.

Assembly language users also use high level languages (HLLs); assembly's most outspoken opponents rarely use anything but HLLs. Who would you believe, an expert well versed in both types of programming languages or someone who has never taken the time to learn assembly language and develop an honest opinion of its capabilities?

In a conversation with someone, I would go to great lengths to address each of the above issues. Indeed, in a rough draft of this chapter I spent about ten pages explaining what is wrong with each of the above statements. However, this book is long enough and I felt that very little was gained by going on and on about these points. Nonetheless, a brief rebuttal to each of the above points is in order, if for no other reason than to keep you from thinking there isn't a decent defense for these statements.

Assembly is hard to learn. So is any language you don't already know. Try learning (really learning) APL, Prolog, or Smalltalk sometime. Once you learn Pascal, learning another language like C, BASIC, FORTRAN, Modula-2, or Ada is fairly easy because these languages are quite similar to Pascal. On the other hand, learning a dissimilar language like Prolog is not so simple. Assembly language is also quite different from Pascal. It will be a little harder to learn than one of the other Pascal-like languages. However, learning assembly isn't much more difficult than learning your first programming language.

Assembly is hard to read and understand. It sure is, if you don't know it. Most people who make this statement simply don't know assembly. Of course, it's very easy to write impossible-to-read assembly language programs. It's also quite easy to write impossible-to-read C, Prolog, and APL programs. With experience, you will find assembly as easy to read as other languages.

Assembly is hard to debug. Same argument as above. If you don't have much experience debugging assembly language programs, it's going to be hard to debug them. Remember what it was like finding bugs in your first Pascal (or other HLL) programs? Anytime you learn a new programming language you'll have problems debugging programs in that language until you gain experience.

Assembly is hard to maintain. C programs are hard to maintain. Indeed, *programs* are hard to maintain period. Inexperienced assembly language programmers tend to write hard to maintain programs. Writing maintainable programs isn't a talent. It's a skill you develop through experience.

Assembly language is hard. This statement actually has a ring of truth to it. For the longest time assembly language programmers wrote their programs completely from scratch, often "re-inventing the wheel." HLL programmers, especially C, Ada, and Modula-2 programmers, have long enjoyed the benefits of a *standard library* package which solves many common programming problems. Assembly language programmers, on the other hand, have been known to rewrite an integer output routine every time they need one. This book does *not* take that approach. Instead, it takes advantage of some work done at the University of California, Riverside: the *UCR Standard Library for 80x86 Assembly Language Programmers.* These subroutines simplify assembly language just as the C standard library aids C programmers. The library source listings are available electronically via Internet and various other communication services as well as on a companion diskette.

Assembly language programming is time consuming. Software engineers estimate that developers spend only about thirty percent of their time coding a solution to a problem. Even if it took twice as

much time to write a program in assembly versus some HLL, there would only be a fifteen percent difference in the total project completion time. In fact, *good* assembly language programmers do not need twice as much time to implement something in assembly language. It is true using a HLL will save *some* time; however, the savings is insufficient to counter the benefits of using assembly language.

Improved compiler technology has eliminated the need for assembly language. This isn't true and probably never will be true. Optimizing compilers are getting better every day. However, assembly language programmers get better performance by writing their code *differently* than they would if they were using some HLL. If assembly language programmers wrote their programs in C and then translated them manually into assembly, a good C compiler would produce equivalent, or even better, code. Those who make this claim about compiler technology are comparing their *hand*-compiled code against that produced by a compiler. Compilers do a much better job of compiling than humans. Then again, you'll never catch an assembly language programmer writing "C code with MOV instructions." After all, that's why you use C compilers.

Today, machines are so fast that we no longer need to use assembly. It is amazing that people will spend lots of money to buy a machine slightly faster than the one they own, but they won't spend any extra time writing their code in assembly so it runs faster on the same hardware. There are many raging debates about the speed of machines versus the speed of the software, but one fact remains: users always want more speed. On any given machine, the fastest possible programs will be written in assembly language[2].

If you need more speed, you should use a better algorithm rather than switch to assembly language. Why can't you use this better algorithm in assembly language? What if you're already using the best algorithm you can find and it's still too slow? This is a totally bogus argument against assembly language. Any algorithm you can implement in a HLL you can implement in assembly. On the other hand, there are many algorithms you can implement in assembly which you cannot implement in a HLL[3].

Machines have so much memory today, saving space using assembly is not important. If you give someone an inch, they'll take a mile. Nowhere in programming does this saying have more application than in program memory use. For the longest time, programmers were quite happy with 4 Kbytes. Later, machines had 32 or even 64 Kilobytes. The programs filled up memory accordingly. Today, many machines have 32 or 64 *megabytes* of memory installed and some applications use it all. There are lots of technical reasons why programmers should strive to write shorter programs, though now is not the time to go into that. Let's just say that space *is* important and programmers should strive to write programs as short as possible regardless of how much main memory they have in their machine.

Assembly language is not portable. This is an undeniable fact. An 80x86 assembly language program written for an IBM PC will not run on an Apple Macintosh[4]. Indeed, assembly language programs written for the Apple Macintosh will not run on an Amiga, even though they share the same 680x0 microprocessor. If you need to run your program on different machines, you'll have to think long and hard about using assembly language. Using C (or some other HLL) is no guarantee that your program will be portable. C programs written for the IBM PC won't compile and run on a Macintosh. And even if they did, most Mac owners wouldn't accept the result.

Portability is probably the biggest complaint people have against assembly language. They refuse to use assembly because it is not portable, and then they turn around and write equally non-portable programs in C.

Yes, there are lots of lies, misconceptions, myths, and half-truths concerning assembly language. Whatever you do, make sure you learn assembly language before forming your own opinions[5]. Speaking

2. That is not to imply that assembly language programs are always faster than HLL programs. A poorly written assembly language program can run much slower than an equivalent HLL program. On the other hand, if a program is written in an HLL it is certainly possible to write a faster one in assembly.

3. We'll see some of these algorithms later in the book. They deal with instruction sequencing and other tricks based on how the processor operates.

4. Strictly speaking, this is not true. There is a program called SoftPC which emulates an IBM PC using an 80286 *interpreter*. However, 80x86 assembly language programs will not run in native mode on the Mac's 680x0 microprocessor.

out in ignorance may impress others who know less than you do, but it won't impress those who know the truth.

2 What's Right With Assembly Language?

An old joke goes something like this: "There are three reasons for using assembly language: speed, speed, and more speed." Even those who absolutely hate assembly language will admit that if speed is your primary concern, assembly language is the way to go. Assembly language has several benefits:

- Speed. Assembly language programs are generally the fastest programs around.
- Space. Assembly language programs are often the smallest.
- Capability. You can do things in assembly which are difficult or impossible in HLLs.
- Knowledge. Your knowledge of assembly language will help you write better programs, even when using HLLs.

Assembly language is the uncontested speed champion among programming languages. An expert assembly language programmer will almost always produce a faster program than an expert C programmer[6]. While certain programs may not benefit much from implementation in assembly, you can speed up many programs by a factor of five or ten over their HLL counterparts by careful coding in assembly language; even greater improvement is possible if you're not using an optimizing compiler. Alas, speedups on the order of five to ten times are generally not achieved by beginning assembly language programmers. However, if you spend the time to learn assembly language really well, you too can achieve these impressive performance gains.

Despite some people's claims that programmers no longer have to worry about memory constraints, there are many programmers who need to write smaller programs. Assembly language programs are often less than one-half the size of comparable HLL programs. This is especially impressive when you consider the fact that data items generally consume the same amount of space in both types of programs, and that data is responsible for a good amount of the space used by a typical application. Saving space saves money. Pure and simple. If a program requires 1.5 megabytes, it will not fit on a 1.44 Mbyte floppy. Likewise, if an application requires 2 megabytes RAM, the user will have to install an extra megabyte if there is only one available in the machine[7]. Even on big machines with 32 or more megabytes, writing gigantic applications isn't excusable. Most users put more than eight megabytes in their machines so they can run *multiple* programs from memory at one time. The bigger a program is, the fewer applications will be able to coexist in memory with it. Virtual memory isn't a particularly attractive solution either. With virtual memory, the bigger an application is, the slower the system will run as a result of that program's size.

Capability is another reason people resort to assembly language. HLLs are an abstraction of a typical machine architecture. They are designed to be independent of the particular machine architecture. As a result, they rarely take into account any special features of the machine, features which are available to assembly language programmers. If you want to use such features, you will need to use assembly language. A really good example is the input/output instructions available on the 80x86 microprocessors. These instructions let you directly access certain I/O devices on the computer. In general, such access is not part of any high level language. Indeed, some languages like C pride themselves on not supporting

5. Alas, a typical ten-week course is rarely sufficient to learn assembly language well enough to develop an informed opinion on the subject. Probably three months of eight-hour days using the stuff would elevate you to the point where you could begin to make some informed statements on the subject. Most people wouldn't be able to consider themselves "good" at assembly language programs until they've been using the stuff for at least a year.

6. There is absolutely no reason why an assembly language programmer would produce a *slower* program since that programmer could look at the output of the C compiler and copy whatever code runs faster than the hand produced code. HLL programmers don't have an equivalent option.

7. You can substitute any numbers here you like. One fact remains though, programmers are famous for assuming users have more memory than they really do.

any specific I/O operations[8]. In assembly language you have no such restrictions. Anything you can do on the machine you can do in assembly language. This is definitely *not* the case with most HLLs.

Of course, another reason for learning assembly language is just for the knowledge. Now some of you may be thinking, "Gee, that would be wonderful, but I've got lots to do. My time would be better spent writing code than learning assembly language." There are some practical reasons for learning assembly, even if you never intend to write a single line of assembly code. If you know assembly language well, you'll have an appreciation for the compiler, and you'll know exactly what the compiler is doing with all those HLL statements. Once you see how compilers translate seemingly innocuous statements into a ton of machine code, you'll want to search for better ways to accomplish the same thing. Good assembly language programmers make better HLL programmers because they understand the limitations of the compiler and they know what it's doing with their code. Those who don't know assembly language will accept the poor performance their compiler produces and simply shrug it off.

Yes, assembly language is definitely worth the effort. The only scary thing is that once you learn it really well, you'll probably start using it far more than you ever dreamed you would. That is a common malady among assembly language programmers. Seems they can't stand what the compilers are doing with their programs.

3 Organization of This Text and Pedagogical Concerns

This book is divided into seven main sections: a section on machine organization and architecture, a section on basic assembly language, a section on intermediate assembly language, a section on interrupts and resident programs, a section covering IBM PC hardware peculiarities, a section on optimization, and various appendices. It is doubtful that any single (even year-long) college course could cover all this material, the final chapters were included to support compiler design, microcomputer design, operating systems, and other courses often found in a typical CS program.

Developing a text such as this one is a very difficult task. First of all, different universities have different ideas about how this course should be taught. Furthermore, different schools spend differing amounts of time on this subject (one or two quarters, a semester, or even a year). Furthermore, different schools cover different material in the course. For example, some schools teach a "Machine Organization" course that emphasizes hardware concepts and presents the assembly language instruction set, but does not expect students to write real assembly language programs (that's the job of a compiler). Other schools teach a "Machine Organization and Assembly Language" course that combines hardware and software issues together into one course. Still others teach a "Machine Organization" or "Digital Logic" course as a prerequisite to an "Assembly Language" course. Still others teach "Assembly Language Programming" as a course and leave the hardware for a "Computer Architecture" course later in the curriculum. Finally, let us not forget that some people will pick up this text and use it to learn machine organization or assembly language programming on their own, without taking a formal course on the subject. A good *textbook* in this subject area must be adaptable to the needs of the course, instructor, and student. These requirements place enough demands on an author, but I wanted more for this text. Many textbooks teach a particular subject well, but once you've read and understood them, they do not serve well as a reference guide. Given the cost of textbooks today, it is a real shame that many textbooks' value diminishes once the course is complete. I sought to create a textbook that will explain many difficult concepts in as friendly a manner as possible *and* will serve as a reference guide once you've mastered the topic. By moving advanced material you probably won't cover in a typical college course into later chapters and by organizing this text so you can continue using it once the course is over, I hope to provide you with an excellent value in this text.

Since this volume attempts to satisfy the requirements of several different courses, as well as provide an excellent reference, you will probably find that it contains far more material than any single course

8. Certain languages on the PC support extensions to access the I/O devices since this is such an obvious limitation of the language. However, such extensions are not part of the actual language.

would actually cover. For example, the first section of this text covers machine organization. If you've already covered this material in a previous course, your instructor may elect to skip the first four chapters or so. For those courses that teach only assembly language, the instructor may decide to skip chapters two and three. Schools operating on a ten-week quarter system may cover the material in each chapter only briefly (about one week per chapter). Other schools may cover the material in much greater depth because they have more time.

When writing this text, I choose to pick a subject and cover it in depth before proceeding to the next topic. This pedagogy (teaching method) is unusual. Most assembly language texts jump around to different topics, lightly touching on each one and returning to them as further explanation is necessary. Unfortunately, such texts make poor references; trying to lookup information in such a book is difficult, at best, because the information is spread throughout the book. Since I want this text to serve as a reasonable reference manual, such an organization was unappealing.

The problem with a straight reference manual is three-fold. First, reference manuals are often organized in a manner that makes it easy to look something up, not in a logical order that makes the material easy to learn. For example, most assembly language reference manuals introduce the instruction set in alphabetical order. However, you do not learn the instruction set in this manner. The second problem with a (good) reference manual is that it presents the material in far greater depth than most beginners can handle; this is why most texts keep returning to a subject, they add a little more depth on each return to the subject. Finally, reference texts can present material in any order. The author need not ensure that a discussion only include material appearing earlier in the text. Material in the early chapters of a reference manual can refer to later chapters; a typical college textbook should *not* do this.

To receive maximum benefit from this text, you need to read it understanding its organization. This is *not* a text you read from front to back, making sure you understand each and every little detail before proceeding to the next. I've covered many topics in this text in considerable detail. Someone learning assembly language for the first time will become overwhelmed with the material that appears in each chapter. Typically, you will read over a chapter once to learn the basic essentials and then refer back to each chapter learning additional material as you need it. Since it is unlikely that you will know which material is basic or advanced, I've taken the liberty of describing which sections are basic, intermediate, or advanced at the beginning of each chapter. A ten-week course, covering this entire text for example, might only deal with the basic topics. In a semester course, there is time to cover the intermediate material as well. Depending on prerequisites and length of course, the instructor can elect to teach this material at any level of detail (or even jump around in the text).

In the past, if a student left an assembly language class and could actually implement an algorithm in assembly language, the instructor probably considered the course a success. However, compiler technology has progressed to the point that simply "getting something to work" in assembly language is pure folly. If you don't write your code efficiently in assembly language, you may as well stick with HLLs. They're easy to use, and the compiler will probably generate faster code than you if you're careless in the coding process.

This text spends a great deal of time on machine and data organization. There are two important reasons for this. First of all, to write efficient code on modern day processors requires an intimate knowledge of what's going on in the hardware. Without this knowledge, your programs on the 80486 and later could run at less than half their possible speed. To write the best possible assembly language programs you must be familiar with how the hardware operates. Another reason this text emphasizes computer organization is that most colleges and universities are more interested in teaching machine organization than they are a particular assembly language. While the typical college student won't have much need for assembly language during the four years as an undergraduate, the machine organization portion of the class is useful in several upper division classes. Classes like data structures and algorithms, computer architecture, operating systems, programming language design, and compilers all benefit from an introductory course in computer organization. That's why this text devotes an entire section to that subject.

4 Obtaining Program Source Listings and Other Materials in This Text

All of the software appearing in this text is available on the companion diskette. The material for this text comes in two parts: source listings of various examples presented in this text and the code for the *UCR Standard Library for 80x86 Assembly Language Programmers*. The UCR Standard Library is also available electronically from several different sources (including Internet, BIX, and other on-line services).

You may obtain the files electronically via ftp from the following Internet address:

ftp.cs.ucr.edu

Log onto ftp.cs.ucr.edu using the anonymous account name and any password. Switch to the "/pub/pc/ibmpcdir" subdirectory (this is UNIX so make sure you use lowercase letters). You will find the appropriate files by searching through this directory.

The exact filename(s) of this material may change with time, and different services use different names for these files. Generally posting a message enquiring about the UCR Standard Library or this text will generate appropriate responses.

Data Representation Chapter One

Probably the biggest stumbling block most beginners encounter when attempting to learn assembly language is the common use of the binary and hexadecimal numbering systems. Many programmers think that hexadecimal (or hex[1]) numbers represent absolute proof that God never intended anyone to work in assembly language. While it is true that hexadecimal numbers are a little different from what you may be used to, their advantages outweigh their disadvantages by a large margin. Nevertheless, understanding these numbering systems is important because their use simplifies other complex topics including boolean algebra and logic design, signed numeric representation, character codes, and packed data.

1.0 Chapter Overview

This chapter discusses several important concepts including the binary and hexadecimal numbering systems, binary data organization (bits, nibbles, bytes, words, and double words), signed and unsigned numbering systems, arithmetic, logical, shift, and rotate operations on binary values, bit fields and packed data, and the ASCII character set. This is basic material and the remainder of this text depends upon your understanding of these concepts. If you are already familiar with these terms from other courses or study, you should at least skim this material before proceeding to the next chapter. If you are unfamiliar with this material, or only vaguely familiar with it, you should study it carefully before proceeding. *All of the material in this chapter is important!* Do not skip over any material.

1.1 Numbering Systems

Most modern computer systems do not represent numeric values using the decimal system. Instead, they typically use a binary or two's complement numbering system. To understand the limitations of computer arithmetic, you must understand how computers represent numbers.

1.1.1 A Review of the Decimal System

You've been using the decimal (base 10) numbering system for so long that you probably take it for granted. When you see a number like "123", you don't think about the value 123; rather, you generate a mental image of how many items this value represents. In reality, however, the number 123 represents:

$$1*10^2 + 2 * 10^1 + 3*10^0$$

or

$$100+20+3$$

Each digit appearing to the left of the decimal point represents a value between zero and nine times an increasing power of ten. Digits appearing to the right of the decimal point represent a value between zero and nine times an increasing negative power of ten. For example, the value 123.456 means:

$$1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$$

or

1. Hexadecimal is often abbreviated as *hex* even though, technically speaking, hex means base six, not base sixteen.

$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

1.1.2 The Binary Numbering System

Most modern computer systems (including the IBM PC) operate using binary logic. The computer represents values using two voltage levels (usually 0v and +5v). With two such levels we can represent exactly two different values. These could be any two different values, but by convention we use the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary numbering system. Since there is a correspondence between the logic levels used by the 80x86 and the two digits used in the binary numbering system, it should come as no surprise that the IBM PC employs the binary numbering system.

The binary numbering system works just like the decimal numbering system, with two exceptions: binary only allows the digits 0 and 1 (rather than 0-9), and binary uses powers of two rather than powers of ten. Therefore, it is very easy to convert a binary number to decimal. For each "1" in the binary string, add in 2^n where "n" is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents:

$$1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$$
$$=$$
$$128 + 64 + 8 + 2$$
$$=$$
$$202_{10}$$

To convert decimal to binary is slightly more difficult. You must find those powers of two which, when added together, produce the decimal result. The easiest method is to work from the a large power of two down to 2^0. Consider the decimal value 1359:

- 2^{10} =1024, 2^{11}=2048. So 1024 is the largest power of two less than 1359. Subtract 1024 from 1359 and begin the binary value on the left with a "1" digit. Binary = "1", Decimal result is 1359 - 1024 = 335.
- The next lower power of two (2^9 = 512) is greater than the result from above, so add a "0" to the end of the binary string. Binary = "10", Decimal result is still 335.
- The next lower power of two is 256 (2^8). Subtract this from 335 and add a "1" digit to the end of the binary number. Binary = "101", Decimal result is 79.
- 128 (2^7) is greater than 79, so tack a "0" to the end of the binary string. Binary = "1010", Decimal result remains 79.
- The next lower power of two (2^6 = 64) is less than79, so subtract 64 and append a "1" to the end of the binary string. Binary = "10101", Decimal result is 15.
- 15 is less than the next power of two (2^5 = 32) so simply add a "0" to the end of the binary string. Binary = "101010", Decimal result is still 15.
- 16 (2^4) is greater than the remainder so far, so append a "0" to the end of the binary string. Binary = "1010100", Decimal result is 15.
- 2^3 (eight) is less than 15, so stick another "1" digit on the end of the binary string. Binary = "10101001", Decimal result is 7.
- 2^2 is less than seven, so subtract four from seven and append another one to the binary string. Binary = "101010011", decimal result is 3.
- 2^1 is less than three, so append a one to the end of the binary string and subtract two from the decimal value. Binary = "1010100111", Decimal result is now 1.
- Finally, the decimal result is one, which is 2^0, so add a final "1" to the end of the binary string. The final binary result is "10101001111"

Binary numbers, although they have little importance in high level languages, appear everywhere in assembly language programs.

1.1.3 Binary Formats

In the purest sense, every binary number contains an infinite number of digits (or *bits* which is short for binary digits). For example, we can represent the number five by:

101 00000101 0000000000101 ... 000000000000101

Any number of leading zero bits may precede the binary number without changing its value.

We will adopt the convention ignoring any leading zeros. For example, 101_2 represents the number five. Since the 80x86 works with groups of eight bits, we'll find it much easier to zero extend all binary numbers to some multiple of four or eight bits. Therefore, following this convention, we'd represent the number five as 0101_2 or 00000101_2.

In the United States, most people separate every three digits with a comma to make larger numbers easier to read. For example, 1,023,435,208 is much easier to read and comprehend than 1023435208. We'll adopt a similar convention in this text for binary numbers. We will separate each group of four binary bits with a space. For example, the binary value 1010111110110010 will be written 1010 1111 1011 0010.

We often pack several values together into the same binary number. One form of the 80x86 MOV instruction (see appendix D) uses the binary encoding 1011 0rrr dddd dddd to pack three items into 16 bits: a five-bit operation code (10110), a three-bit register field (rrr), and an eight-bit immediate value (dddd dddd). For convenience, we'll assign a numeric value to each bit position. We'll number each bit as follows:

1) The rightmost bit in a binary number is bit position zero.
2) Each bit to the left is given the next successive bit number.

An eight-bit binary value uses bits zero through seven:

$$X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

A 16-bit binary value uses bit positions zero through fifteen:

$$X_{15}\ X_{14}\ X_{13}\ X_{12}\ X_{11}\ X_{10}\ X_9\ X_8\ X_7\ X_6\ X_5\ X_4\ X_3\ X_2\ X_1\ X_0$$

Bit zero is usually referred to as the *low order* (L.O.) bit. The left-most bit is typically called the *high order* (H.O.) bit. We'll refer to the intermediate bits by their respective bit numbers.

1.2 Data Organization

In pure mathematics a value may take an arbitrary number of bits. Computers, on the other hand, generally work with some specific number of bits. Common collections are single bits, groups of four bits (called *nibbles*), groups of eight bits (called *bytes*), groups of 16 bits (called *words*), and more. The sizes are not arbitrary. There is a good reason for these particular values. This section will describe the bit groups commonly used on the Intel 80x86 chips.

1.2.1 Bits

The smallest "unit" of data on a binary computer is a single *bit*. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit.

With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, you are *not* limited to representing binary data types (that is, those objects which have only two distinct values). You could use a single bit to represent the numbers 723 and 1,245. Or perhaps 6,254 and 5. You could also use a single bit to represent the colors red and blue. You could even represent two unrelated objects with a single bit,. For example, you could represent the color red and the number 3,256 with a single bit. You can represent *any* two different values with a single bit. However, you can represent *only* two different values with a single bit.

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the values true and false. How can you tell by looking at the bits? The answer, of course, is that you can't. But this illustrates the whole idea behind computer data structures: *data is what you define it to be.* If you use a bit to represent a boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any true meaning, you must be consistent. That is, if you're using a bit to represent true or false at one point in your program, you shouldn't use the true/false value stored in that bit to represent red or blue later.

Since most items you'll be trying to model require more than two different values, single bit values aren't the most popular data type you'll use. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. However, you will soon see that individual bits are difficult to manipulate, so we'll often use other data types to represent boolean values.

1.2.2 Nibbles

A *nibble* is a collection of four bits. It wouldn't be a particularly interesting data structure except for two items: BCD (*binary coded decimal*) numbers and hexadecimal numbers. It takes four bits to represent a single BCD or hexadecimal digit. With a nibble, we can represent up to 16 distinct values. In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits (see "The Hexadecimal Numbering System" on page 17). BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits. In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

1.2.3 Bytes

Without question, the most important data structure used by the 80x86 microprocessor is the byte. A byte consists of eight bits and is the smallest addressable datum (data item) on the 80x86 microprocessor. Main memory and I/O addresses on the 80x86 are all byte addresses. This means that the smallest item that can be individually accessed by an 80x86 program is an eight-bit value. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits. The bits in a byte are normally numbered from zero to seven using the convention in Figure 1.1.

Bit 0 is the *low order bit* or *least significant bit*, bit 7 is the *high order bit* or *most significant bit* of the byte. We'll refer to all other bits by their number.

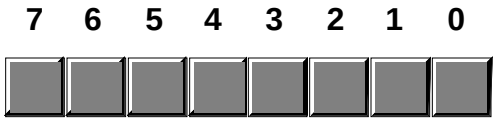


Figure 1.1: Bit Numbering in a Byte

Note that a byte also contains exactly two nibbles (see Figure 1.2).

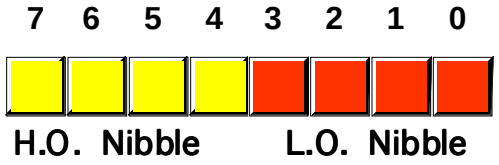


Figure 1.2: The Two Nibbles in a Byte

Bits 0..3 comprise the *low order nibble*, bits 4..7 form the *high order nibble*. Since a byte contains exactly two nibbles, byte values require two hexadecimal digits.

Since a byte contains eight bits, it can represent 2^8, or 256, different values. Generally, we'll use a byte to represent numeric values in the range 0..255, signed numbers in the range -128..+127 (see "Signed and Unsigned Numbers" on page 23), ASCII/IBM character codes, and other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

Since the 80x86 is a byte addressable machine (see "Memory Layout and Access" on page 145), it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, we'll often represent the boolean values true and false by 00000001_2 and 00000000_2 (respectively).

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, the IBM PC uses a variant of the ASCII character set (see "The ASCII Character Set" on page 28). There are 128 defined codes in the ASCII character set. IBM uses the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols. See Appendix A for the character/code assignments.

1.2.4 Words

A word is a group of 16 bits. We'll number the bits in a word starting from zero on up to fifteen. The bit numbering appears in Figure 1.3.

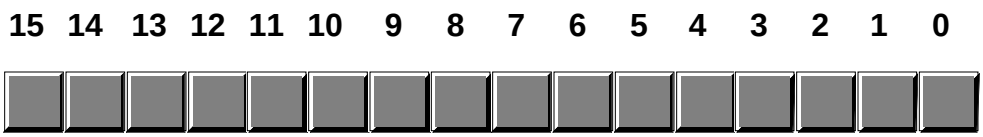


Figure 1.3: Bit Numbers in a Word

Like the byte, bit 0 is the low order bit and bit 15 is the high order bit. When referencing the other bits in a word use their bit position number.

Notice that a word contains exactly two bytes. Bits 0 through 7 form the low order byte, bits 8 through 15 form the high order byte (see Figure 1.4).

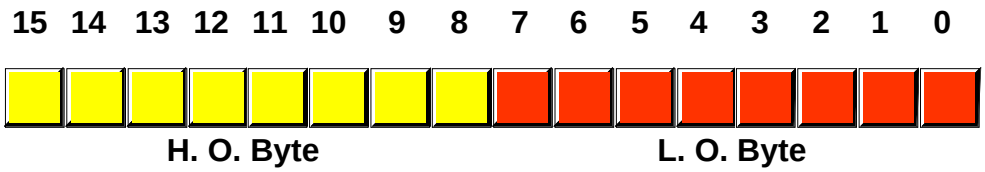


Figure 1.4: The Two Bytes in a Word

Naturally, a word may be further broken down into four nibbles as shown in Figure 1.5.

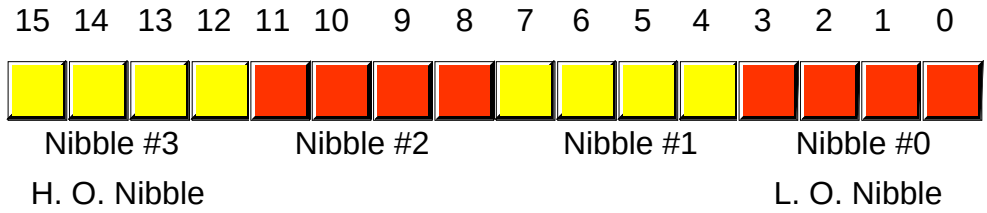


Figure 1.5: Nibbles in a Word

Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. The other two nibbles are “nibble one” or “nibble two”.

With 16 bits, you can represent 2^{16} (65,536) different values. These could be the values in the range 0..65,535 (or, as is usually the case, -32,768..+32,767) or any other data type with no more than 65,536 values. The three major uses for words are integer values, offsets, and segment values (see“Memory Layout and Access” on page 145 for a description of segments and offsets).

Words can represent integer values in the range 0..65,535 or -32,768..32,767. Unsigned numeric values are represented by the binary value corresponding to the bits in the word. Signed numeric values use the two’s complement form for numeric values (see “Signed and Unsigned Numbers” on page 23). Segment values, which are always 16 bits long, constitute the *paragraph address* of a code, data, extra, or stack segment in memory.

1.2.5 Double Words

A double word is exactly what its name implies, a pair of words. Therefore, a double word quantity is 32 bits long as shown in Figure 1.6.

Figure 1.6: Bit Numbers in a Double Word

Naturally, this double word can be divided into a high order word and a low order word, or four different bytes, or eight different nibbles (see Figure 1.7).

Double words can represent all kinds of different things. First and foremost on the list is a segmented address. Another common item represented with a double word is a 32-bit

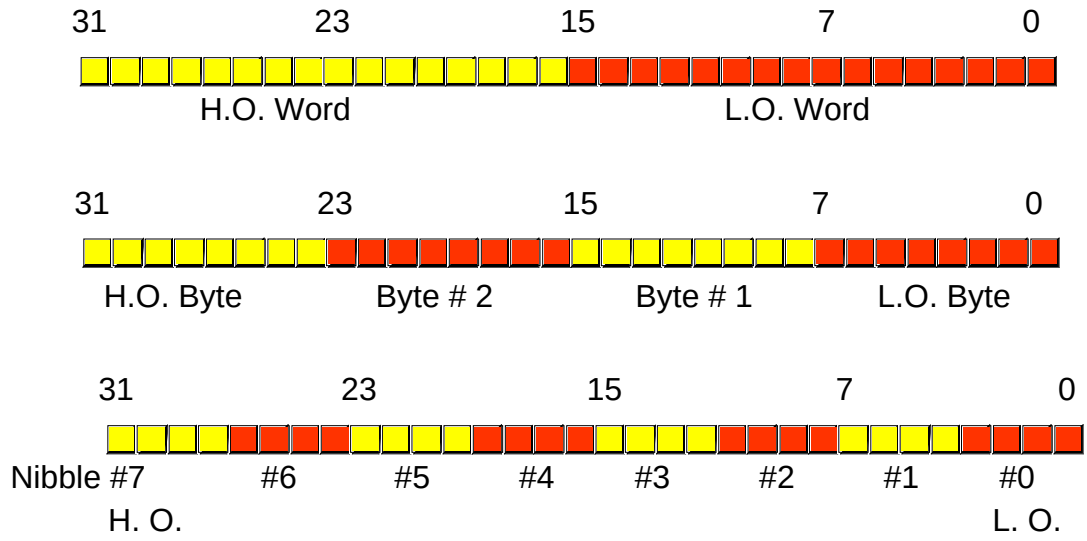


Figure 1.7: Nibbles, Bytes, and Words in a Double Word

integer value (which allows unsigned numbers in the range 0..4,294,967,295 or signed numbers in the range -2,147,483,648..2,147,483,647). 32-bit floating point values also fit into a double word. Most of the time, we'll use double words to hold segmented addresses.

1.3 The Hexadecimal Numbering System

A big problem with the binary system is verbosity. To represent the value 202_{10} requires eight binary digits. The decimal version requires only three decimal digits and, thus, represents numbers much more compactly than does the binary numbering system. This fact was not lost on the engineers who designed binary computer systems. When dealing with large values, binary numbers quickly become too unwieldy. Unfortunately, the computer thinks in binary, so most of the time it is convenient to use the binary numbering system. Although we can convert between decimal and binary, the conversion is not a trivial task. The hexadecimal (base 16) numbering system solves these problems. Hexadecimal numbers offer the two features we're looking for: they're very compact, and it's simple to convert them to binary and vice versa. Because of this, most binary computer systems today use the hexadecimal numbering system[2]. Since the radix (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234_{16} is equal to:

$$1 * 16^3 + 2 * 16^2 + 3 * 16^1 + 4 * 16^0$$

or

$$4096 + 512 + 48 + 4 = 4660_{10}.$$

Each hexadecimal digit can represent one of sixteen values between 0 and 15_{10}. Since there are only ten decimal digits, we need to invent six additional digits to represent the values in the range 10_{10} through 15_{10}. Rather than create new symbols for these digits, we'll use the letters A through F. The following are all examples of valid hexadecimal numbers:

2. Digital Equipment is the only major holdout. They still use octal numbers in most of their systems. A legacy of the days when they produced 12-bit machines.

1234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

Since we'll often need to enter hexadecimal numbers into the computer system, we'll need a different mechanism for representing hexadecimal numbers. After all, on most computer systems you cannot enter a subscript to denote the radix of the associated value. We'll adopt the following conventions:

- All numeric values (regardless of their radix) begin with a decimal digit.
- All hexadecimal values end with the letter "h", e.g., 123A4h[3].
- All binary values end with the letter "b".
- Decimal numbers *may* have a "t" or "d" suffix.

Examples of valid hexadecimal numbers:

1234h 0DEADh 0BEEFh 0AFBh 0FEEDh 0DEAFh

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Consider the following table:

Table 1: Binary/Hex Conversion

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

This table provides all the information you'll ever need to convert any hexadecimal number into a binary number or vice versa.

To convert a hexadecimal number into a binary number, simply substitute the corresponding four bits for each hexadecimal digit in the number. For example, to convert

3. Actually, following hexadecimal values with an "h" is an Intel convention, not a general convention. The 68000 and 65c816 assemblers used in the Macintosh and Apple II denote hexadecimal numbers by prefacing the hex value with a "$" symbol.

0ABCDh into a binary value, simply convert each hexadecimal digit according to the table above:

0	A	B	C	D	Hexadecimal
0000	1010	1011	1100	1101	Binary

To convert a binary number into hexadecimal format is almost as easy. The first step is to pad the binary number with zeros to make sure that there is a multiple of four bits in the number. For example, given the binary number 1011001010, the first step would be to add two bits to the left of the number so that it contains 12 bits. The converted binary value is 001011001010. The next step is to separate the binary value into groups of four bits, e.g., 0010 1100 1010. Finally, look up these binary values in the table above and substitute the appropriate hexadecimal digits, e.g., 2CA. Contrast this with the difficulty of conversion between decimal and binary or decimal and hexadecimal!

Since converting between hexadecimal and binary is an operation you will need to perform over and over again, you should take a few minutes and memorize the table above. Even if you have a calculator that will do the conversion for you, you'll find manual conversion to be a lot faster and more convenient when converting between binary and hex.

1.4 Arithmetic Operations on Binary and Hexadecimal Numbers

There are several operations we can perform on binary and hexadecimal numbers. For example, we can add, subtract, multiply, divide, and perform other arithmetic operations. Although you needn't become an expert at it, you should be able to, in a pinch, perform these operations manually using a piece of paper and a pencil. Having just said that you should be able to perform these operations manually, the correct way to perform such arithmetic operations is to have a calculator which does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Manufacturers of Hexadecimal Calculators:

- Casio
- Hewlett-Packard
- Sharp
- Texas Instruments

This list is, by no means, exhaustive. Other calculator manufacturers probably produce these devices as well. The Hewlett-Packard devices are arguably the best of the bunch . However, they are more expensive than the others. Sharp and Casio produce units which sell for well under $50. If you plan on doing any assembly language programming at all, owning one of these calculators is essential.

Another alternative to purchasing a hexadecimal calculator is to obtain a TSR (Terminate and Stay Resident) program such as SideKick^{tm} which contains a built-in calculator. However, unless you already have one of these programs, or you need some of the other features they offer, such programs are not a particularly good value since they cost more than an actual calculator and are not as convenient to use.

To understand why you should spend the money on a calculator, consider the following arithmetic problem:

```
  9h
+ 1h
----
```

You're probably tempted to write in the answer "10h" as the solution to this problem. But that is not correct! The correct answer is ten, which is "0Ah", not sixteen which is "10h". A similar problem exists with the arithmetic problem:

```
 10h
- 1h
----
```

You're probably tempted to answer "9h" even though the true answer is "0Fh". Remember, this problem is asking "what is the difference between sixteen and one?" The answer, of course, is fifteen which is "0Fh".

Even if the two problems above don't bother you, in a stressful situation your brain will switch back into decimal mode while you're thinking about something else and you'll produce the incorrect result. Moral of the story – if you must do an arithmetic computation using hexadecimal numbers by hand, take your time and be careful about it. Either that, or convert the numbers to decimal, perform the operation in decimal, and convert them back to hexadecimal.

You should never perform binary arithmetic computations. Since binary numbers usually contain long strings of bits, there is too much of an opportunity for you to make a mistake. Always convert binary numbers to hex, perform the operation in hex (preferably with a hex calculator) and convert the result back to binary, if necessary.

1.5 Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND, OR, XOR (exclusive-or), and NOT. Unlike the arithmetic operations, a hexadecimal calculator isn't necessary to perform these operations. It is often easier to do them by hand than to use an electronic device to compute them. The logical AND operation is a dyadic[4] operation (meaning it accepts exactly two operands). These operands are single binary (base 2) bits. The AND operation is:

0 and 0 = 0

0 and 1 = 0

1 and 0 = 0

1 and 1 = 1

A compact way to represent the logical AND operation is with a truth table. A truth table takes the following form:

Table 2: AND Truth Table

AND	0	1
0	0	0
1	0	1

This is just like the multiplication tables you encountered in elementary school. The column on the left and the row at the top represent input values to the AND operation. The value located at the intersection of the row and column (for a particular pair of input values) is the result of logically ANDing those two values together. In English, the logical AND operation is, "If the first operand is one and the second operand is one, the result is one; otherwise the result is zero."

One important fact to note about the logical AND operation is that you can use it to force a zero result. If one of the operands is zero, the result is always zero regardless of the other operand. In the truth table above, for example, the row labelled with a zero input

4. Many texts call this a binary operation. The term dyadic means the same thing and avoids the confusion with the binary numbering system.

contains only zeros and the column labelled with a zero only contains zero results. Conversely, if one operand contains a one, the result is exactly the value of the second operand. These features of the AND operation are very important, particularly when working with bit strings and we want to force individual bits in the string to zero. We will investigate these uses of the logical AND operation in the next section.

The logical OR operation is also a dyadic operation. Its definition is:

0 or 0 = 0

0 or 1 = 1

1 or 0 = 1

1 or 1 = 1

The truth table for the OR operation takes the following form:

Table 3: OR Truth Table

OR	0	1
0	0	1
1	1	1

Colloquially, the logical OR operation is, “If the first operand or the second operand (or both) is one, the result is one; otherwise the result is zero.” This is also known as the *inclusive-OR* operation.

If one of the operands to the logical-OR operation is a one, the result is always one regardless of the second operand’s value. If one operand is zero, the result is always the value of the second operand. Like the logical AND operation, this is an important side-effect of the logical-OR operation that will prove quite useful when working with bit strings (see the next section).

Note that there is a difference between this form of the inclusive logical OR operation and the standard English meaning. Consider the phrase “I am going to the store *or* I am going to the park.” Such a statement implies that the speaker is going to the store or to the park but not to both places. Therefore, the English version of logical OR is slightly different than the inclusive-OR operation; indeed, it is closer to the *exclusive-OR* operation.

The logical XOR (exclusive-or) operation is also a dyadic operation. It is defined as follows:

0 xor 0 = 0

0 xor 1 = 1

1 xor 0 = 1

1 xor 1 = 0

The truth table for the XOR operation takes the following form:

Table 4: XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In English, the logical XOR operation is, “If the first operand or the second operand, but not both, is one, the result is one; otherwise the result is zero.” Note that the exclusive-or operation is closer to the English meaning of the word “or” than is the logical OR operation.

If one of the operands to the logical exclusive-OR operation is a one, the result is always the *inverse* of the other operand; that is, if one operand is one, the result is zero if the other operand is one and the result is one if the other operand is zero. If the first operand contains a zero, then the result is exactly the value of the second operand. This feature lets you selectively invert bits in a bit string.

The logical NOT operation is a monadic[5] operation (meaning it accepts only one operand). It is:

NOT 0 = 1

NOT 1 = 0

The truth table for the NOT operation takes the following form:

Table 5: NOT Truth Table

NOT	0	1
	1	0

1.6 Logical Operations on Binary Numbers and Bit Strings

As described in the previous section, the logical functions work only with single bit operands. Since the 80x86 uses groups of eight, sixteen, or thirty-two bits, we need to extend the definition of these functions to deal with more than two bits. Logical functions on the 80x86 operate on a *bit-by-bit* (or *bitwise*) basis. Given two values, these functions operate on bit zero producing bit zero of the result. They operate on bit one of the input values producing bit one of the result, etc. For example, if you want to compute the logical AND of the following two eight-bit numbers, you would perform the logical AND operation on each column independently of the others:

```
1011 0101
1110 1110
---------
1010 0100
```

This bit-by-bit form of execution can be easily applied to the other logical operations as well.

Since we’ve defined logical operations in terms of binary values, you’ll find it much easier to perform logical operations on binary values than on values in other bases. Therefore, if you want to perform a logical operation on two hexadecimal numbers, you should convert them to binary first. This applies to most of the basic logical operations on binary numbers (e.g., AND, OR, XOR, etc.).

The ability to force bits to zero or one using the logical AND/OR operations and the ability to invert bits using the logical XOR operation is very important when working with strings of bits (e.g., binary numbers). These operations let you selectively manipulate certain bits within some value while leaving other bits unaffected. For example, if you have an eight-bit binary value ‘X’ and you want to guarantee that bits four through seven contain zeros, you could logically AND the value ‘X’ with the binary value 0000 1111. This

5. Monadic means the operator has one operand.

bitwise logical AND operation would force the H.O. four bits to zero and pass the L.O. four bits of 'X' through unchanged. Likewise, you could force the L.O. bit of 'X' to one and invert bit number two of 'X' by logically ORing 'X' with 0000 0001 and logically exclusive-ORing 'X' with 0000 0100, respectively. Using the logical AND, OR, and XOR operations to manipulate bit strings in this fashion is know as *masking* bit strings. We use the term *masking* because we can use certain values (one for AND, zero for OR/XOR) to 'mask out' certain bits from the operation when forcing bits to zero, one, or their inverse.

1.7 Signed and Unsigned Numbers

So far, we've treated binary numbers as unsigned values. The binary number ...00000 represents zero, ...00001 represents one, ...00010 represents two, and so on toward infinity. What about negative numbers? Signed values have been tossed around in previous sections and we've mentioned the two's complement numbering system, but we haven't discussed how to represent negative numbers using the binary numbering system. That is what this section is all about!

To represent signed numbers using the binary numbering system we have to place a restriction on our numbers: they must have a finite and fixed number of bits. As far as the 80x86 goes, this isn't too much of a restriction, after all, the 80x86 can only address a finite number of bits. For our purposes, we're going to severely limit the number of bits to eight, 16, 32, or some other small number of bits.

With a fixed number of bits we can only represent a certain number of objects. For example, with eight bits we can only represent 256 different objects. Negative values are objects in their own right, just like positive numbers. Therefore, we'll have to use some of the 256 different values to represent negative numbers. In other words, we've got to use up some of the positive numbers to represent negative numbers. To make things fair, we'll assign half of the possible combinations to the negative values and half to the positive values. So we can represent the negative values -128..-1 and the positive values 0..127 with a single eight bit byte[6]. With a 16-bit word we can represent values in the range -32,768..+32,767. With a 32-bit double word we can represent values in the range -2,147,483,648..+2,147,483,647. In general, with *n* bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1}-1$.

Okay, so we can represent negative values. Exactly how do we do it? Well, there are many ways, but the 80x86 microprocessor uses the two's complement notation. In the two's complement system, the H.O. bit of a number is a *sign bit*. If the H.O. bit is zero, the number is positive; if the H.O. bit is one, the number is negative. Examples:

For 16-bit numbers:

8000h is negative because the H.O. bit is one.

100h is positive because the H.O. bit is zero.

7FFFh is positive.

0FFFFh is negative.

0FFFh is positive.

If the H.O. bit is zero, then the number is positive and is stored as a standard binary value. If the H.O. bit is one, then the number is negative and is stored in the two's complement form. To convert a positive number to its negative, two's complement form, you use the following algorithm:

1) Invert all the bits in the number, i.e., apply the logical NOT function.

6. Technically, zero is neither positive nor negative. For technical reasons (due to the hardware involved), we'll lump zero in with the positive numbers.

2) Add one to the inverted result.

For example, to compute the eight bit equivalent of -5:

```
0000 0101           Five (in binary).
1111 1010           Invert all the bits.
1111 1011           Add one to obtain result.
```

If we take minus five and perform the two's complement operation on it, we get our original value, 00000101, back again, just as we expect:

```
1111 1011           Two's complement for -5.
0000 0100           Invert all the bits.
0000 0101           Add one to obtain result (+5).
```

The following examples provide some positive and negative 16-bit signed values:

7FFFh: +32767, the largest 16-bit positive number.

8000h: -32768, the smallest 16-bit negative number.

4000h: +16,384.

To convert the numbers above to their negative counterpart (i.e., to negate them), do the following:

```
7FFFh:      0111 1111 1111 1111         +32,767t
            1000 0000 0000 0000         Invert all the bits (8000h)
            1000 0000 0000 0001         Add one (8001h or -32,767t)

8000h:      1000 0000 0000 0000         -32,768t
            0111 1111 1111 1111         Invert all the bits (7FFFh)
            1000 0000 0000 0000         Add one (8000h or -32768t)

4000h:      0100 0000 0000 0000         16,384t
            1011 1111 1111 1111         Invert all the bits (BFFFh)
            1100 0000 0000 0000         Add one (0C000h or -16,384t)
```

8000h inverted becomes 7FFFh. After adding one we obtain 8000h! Wait, what's going on here? -(-32,768) is -32,768? Of course not. But the value +32,768 cannot be represented with a 16-bit signed number, so we cannot negate the smallest negative value. If you attempt this operation, the 80x86 microprocessor will complain about signed arithmetic overflow.

Why bother with such a miserable numbering system? Why not use the H.O. bit as a sign flag, storing the positive equivalent of the number in the remaining bits? The answer lies in the hardware. As it turns out, negating values is the only tedious job. With the two's complement system, most other operations are as easy as the binary system. For example, suppose you were to perform the addition 5+(-5). The result is zero. Consider what happens when we add these two values in the two's complement system:

```
  00000101
  11111011
  --------
1 00000000
```

We end up with a carry into the ninth bit and all other bits are zero. As it turns out, if we ignore the carry out of the H.O. bit, adding two signed values always produces the correct result when using the two's complement numbering system. This means we can use the same hardware for signed and unsigned addition and subtraction. This wouldn't be the case with some other numbering systems.

Except for the questions at the end of this chapter, you will not need to perform the two's complement operation by hand. The 80x86 microprocessor provides an instruction, NEG (negate), which performs this operation for you. Furthermore, all the hexadecimal

calculators will perform this operation by pressing the change sign key (+/- or CHS). Nevertheless, performing a two's complement by hand is easy, and you should know how to do it.

Once again, you should note that the data represented by a set of binary bits depends entirely on the context. The eight bit binary value 11000000b could represent an IBM / ASCII character, it could represent the unsigned decimal value 192, or it could represent the signed decimal value -64, etc. As the programmer, it is your responsibility to use this data consistently.

1.8 Sign and Zero Extension

Since two's complement format integers have a fixed length, a small problem develops. What happens if you need to convert an eight bit two's complement value to 16 bits? This problem, and its converse (converting a 16 bit value to eight bits) can be accomplished via *sign extension* and *contraction* operations. Likewise, the 80x86 works with fixed length values, even when processing unsigned binary numbers. *Zero extension* lets you convert small unsigned values to larger unsigned values.

Consider the value "-64". The eight bit two's complement value for this number is 0C0h. The 16-bit equivalent of this number is 0FFC0h. Now consider the value "+64". The eight and 16 bit versions of this value are 40h and 0040h. The difference between the eight and 16 bit numbers can be described by the rule: "If the number is negative, the H.O. byte of the 16 bit number contains 0FFh; if the number is positive, the H.O. byte of the 16 bit quantity is zero."

To sign extend a value from some number of bits to a greater number of bits is easy, just copy the sign bit into all the additional bits in the new format. For example, to sign extend an eight bit number to a 16 bit number, simply copy bit seven of the eight bit number into bits 8..15 of the 16 bit number. To sign extend a 16 bit number to a double word, simply copy bit 15 into bits 16..31 of the double word.

Sign extension is required when manipulating signed values of varying lengths. Often you'll need to add a byte quantity to a word quantity. You must sign extend the byte quantity to a word before the operation takes place. Other operations (multiplication and division, in particular) may require a sign extension to 32-bits. You must not sign extend unsigned values.

Examples of sign extension:

```
Eight Bits     Sixteen Bits          Thirty-two Bits

 80h            FF80h                 FFFFFF80h
 28h            0028h                 00000028h
 9Ah            FF9Ah                 FFFFFF9Ah
 7Fh            007Fh                 0000007Fh
 ---            1020h                 00001020h
 ---            8088h                 FFFF8088h
```

To extend an unsigned byte you must zero extend the value. Zero extension is very easy – just store a zero into the H.O. byte(s) of the smaller operand. For example, to zero extend the value 82h to 16-bits you simply add a zero to the H.O. byte yielding 0082h.

```
Eight Bits     Sixteen Bits          Thirty-two Bits

 80h            0080h                 00000080h
 28h            0028h                 00000028h
 9Ah            009Ah                 0000009Ah
 7Fh            007Fh                 0000007Fh
 ---            1020h                 00001020h
 ---            8088h                 00008088h
```

Sign contraction, converting a value with some number of bits to the identical value with a fewer number of bits, is a little more troublesome. Sign extension never fails. Given an *m*-bit signed value you can always convert it to an *n*-bit number (where $n > m$) using

sign extension. Unfortunately, given an *n*-bit number, you cannot always convert it to an *m*-bit number if *m* < *n*. For example, consider the value -448. As a 16-bit hexadecimal number, its representation is 0FE40h. Unfortunately, the magnitude of this number is too great to fit into an eight bit value, so you cannot sign contract it to eight bits. This is an example of an overflow condition that occurs upon conversion.

To properly sign contract one value to another, you must look at the H.O. byte(s) that you want to discard. The H.O. bytes you wish to remove must all contain either zero or 0FFh. If you encounter any other values, you cannot contract it without overflow. Finally, the H.O. bit of your resulting value must match *every* bit you've removed from the number. Examples (16 bits to eight bits):

```
FF80h can be sign contracted to 80h
0040h can be sign contracted to 40h
FE40h cannot be sign contracted to 8 bits.
0100h cannot be sign contracted to 8 bits.
```

1.9 Shifts and Rotates

Another set of logical operations which apply to bit strings are the *shift* and *rotate* operations. These two categories can be further broken down into *left shifts*, *left rotates*, *right shifts*, and *right rotates*. These operations turn out to be extremely useful to assembly language programmers.

The left shift operation moves each bit in a bit string one position to the left (see Figure 1.8).

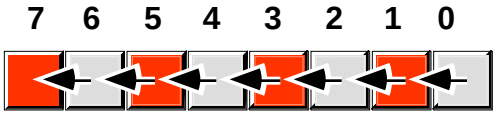


Figure 1.8: Shift Left Operation

Bit zero moves into bit position one, the previous value in bit position one moves into bit position two, etc. There are, of course, two questions that naturally arise: "What goes into bit zero?" and "Where does bit seven wind up?" Well, that depends on the context. We'll shift the value zero into the L.O. bit, and the previous value of bit seven will be the *carry* out of this operation.

Note that shifting a value to the left is the same thing as multiplying it by its radix. For example, shifting a decimal number one position to the left (adding a zero to the right of the number) effectively multiplies it by ten (the radix):

```
1234 SHL 1 = 12340        (SHL 1 = shift left one position)
```

Since the radix of a binary number is two, shifting it left multiplies it by two. If you shift a binary value to the left twice, you multiply it by two twice (i.e., you multiply it by four). If you shift a binary value to the left three times, you multiply it by eight (2*2*2). In general, if you shift a value to the left *n* times, you multiply that value by 2^n.

A right shift operation works the same way, except we're moving the data in the opposite direction. Bit seven moves into bit six, bit six moves into bit five, bit five moves into bit four, etc. During a right shift, we'll move a zero into bit seven, and bit zero will be the carry out of the operation (see Figure 1.9).

Since a left shift is equivalent to a multiplication by two, it should come as no surprise that a right shift is roughly comparable to a division by two (or, in general, a division by the radix of the number). If you perform *n* right shifts, you will divide that number by 2^n.

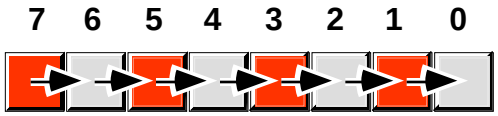


Figure 1.9: Shift Right Operation

There is one problem with shift rights with respect to division: as described above a shift right is only equivalent to an *unsigned* division by two. For example, if you shift the unsigned representation of 254 (0FEh) one place to the right, you get 127 (07Fh), exactly what you would expect. However, if you shift the binary representation of -2 (0FEh) to the right one position, you get 127 (07Fh), which is *not* correct. This problem occurs because we're shifting a zero into bit seven. If bit seven previously contained a one, we're changing it from a negative to a positive number. Not a good thing when dividing by two.

To use the shift right as a division operator, we must define a third shift operation: *arithmetic shift right*[7]. An arithmetic shift right works just like the normal shift right operation (a *logical shift right*) with one exception: instead of shifting a zero into bit seven, an arithmetic shift right operation leaves bit seven alone, that is, during the shift operation it does not modify the value of bit seven as Figure 1.10 shows.

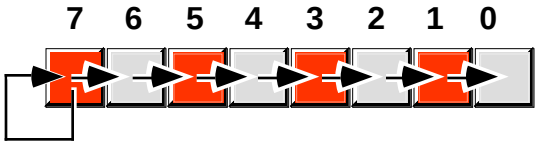


Figure 1.10: Arithmetic Shift Right Operation

This generally produces the result you expect. For example, if you perform the arithmetic shift right operation on -2 (0FEh) you get -1 (0FFh). Keep one thing in mind about arithmetic shift right, however. This operation always rounds the numbers to the closest integer *which is less than or equal to the actual result*. Based on experiences with high level programming languages and the standard rules of integer truncation, most people assume this means that a division always truncates towards zero. But this simply isn't the case. For example, if you apply the arithmetic shift right operation on -1 (0FFh), the result is -1, not zero. -1 is less than zero so the arithmetic shift right operation rounds towards minus one. This is not a "bug" in the arithmetic shift right operation. This is the way integer division typically gets defined. The 80x86 integer division instruction also produces this result.

Another pair of useful operations are *rotate left* and *rotate right*. These operations behave like the shift left and shift right operations with one major difference: the bit shifted out from one end is shifted back in at the other end.

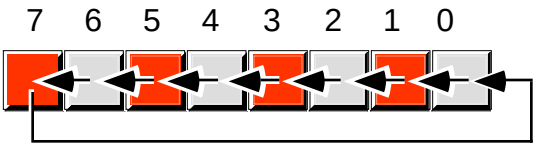


Figure 1.11: Rotate Left Operation

7. There is no need for an arithmetic shift left. The standard shift left operation works for both signed and unsigned numbers, assuming no overflow occurs.

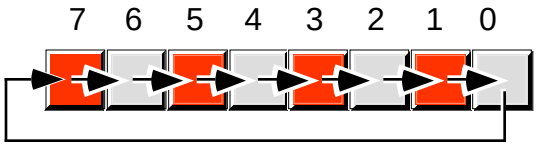


Figure 1.12: Rotate Right Operation

1.10 Bit Fields and Packed Data

Although the 80x86 operates most efficiently on byte, word, and double word data types, occasionally you'll need to work with a data type that uses some number of bits other than eight, 16, or 32. For example, consider a date of the form "4/2/88". It takes three numeric values to represent this date: a month, day, and year value. Months, of course, take on the values 1..12. It will require at least four bits (maximum of sixteen different values) to represent the month. Days range between 1..31. So it will take five bits (maximum of 32 different values) to represent the day entry. The year value, assuming that we're working with values in the range 0..99, requires seven bits (which can be used to represent up to 128 different values). Four plus five plus seven is 16 bits, or two bytes. In other words, we can pack our date data into two bytes rather than the three that would be required if we used a separate byte for each of the month, day, and year values. This saves one byte of memory for each date stored, which could be a substantial saving if you need to store a lot of dates. The bits could be arranged as shown in .

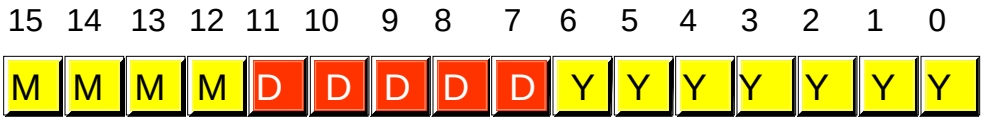


Figure 1.13: Packed Date Format

MMMM represents the four bits making up the month value, DDDDD represents the five bits making up the day, and YYYYYYY is the seven bits comprising the year. Each collection of bits representing a data item is a *bit field*. April 2nd, 1988 would be represented as 4158h:

```
0100 00010 1011000    = 0100 0001 0101 1000b or 4158h
 4     2      88
```

Although packed values are *space efficient* (that is, very efficient in terms of memory usage), they are computationally *inefficient* (slow!). The reason? It takes extra instructions to unpack the data packed into the various bit fields. These extra instructions take additional time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything.

Examples of practical packed data types abound. You could pack eight boolean values into a single byte, you could pack two BCD digits into a byte, etc.

1.11 The ASCII Character Set

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 through

1Fh (31), form a special set of non-printing characters called the control characters. We call them control characters because they perform various printer / display control operations rather than displaying symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters[8], line feed (which moves the cursor down one line on the output device), and back space (which moves the cursor back one position to the left). Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprise various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code 20h) and the numeric digits (ASCII codes 30h..39h). Note that the numeric digits differ from their numeric values only in the H.O. nibble. By subtracting 30h from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters is reserved for the upper case alphabetic characters. The ASCII codes for the characters “A”..”Z” lie in the range 41h..5Ah (65..90). Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes are reserved for the lower case alphabetic symbols, five additional special symbols, and another control character (delete). Note that the lower case character symbols use the ASCII codes 61h..7Ah. If you convert the codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position. For example, consider the character code for “E” and “e” in Figure 1.14.

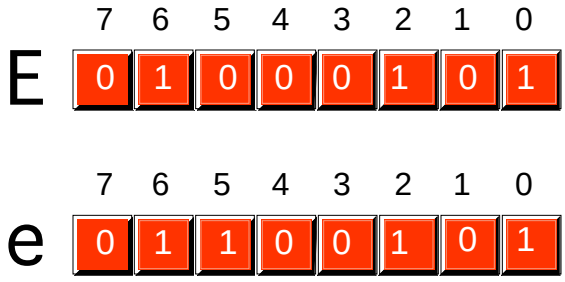


Figure 1.14: ASCII Codes for “E” and “e”.

The only place these two codes differ is in bit five. Upper case characters always contain a zero in bit five; lower case alphabetic characters always contain a one in bit five. You can use this fact to quickly convert between upper and lower case. If you have an upper case character you can force it to lower case by setting bit five to one. If you have a lower case character and you wish to force it to upper case, you can do so by setting bit five to zero. You can toggle an alphabetic character between upper and lower case by simply inverting bit five.

Indeed, bits five and six determine which of the four groups in the ASCII character set you’re in:

8. Historically, carriage return refers to the *paper carriage* used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left hand side of the paper.

Bit 6	Bit 5	Group
0	0	Control Characters
0	1	Digits & Punctuation
1	0	Upper Case & Special
1	1	Lower Case & Special

So you could, for instance, convert any upper or lower case (or corresponding special) character to its equivalent control character by setting bits five and six to zero.

Consider, for a moment, the ASCII codes of the numeric digit characters:

Char Dec Hex

“0”	48	30h
“1”	49	31h
“2”	50	32h
“3”	51	33h
“4”	52	34h
“5”	53	35h
“6”	54	36h
“7”	55	37h
“8”	56	38h
“9”	57	39h

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important – the L.O. nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (i.e., setting to zero) the H.O. nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0..9 to its ASCII character representation by simply setting the H.O. nibble to three. Note that you can use the logical-AND operation to force the H.O. bits to zero; likewise, you can use the logical-OR operation to force the H.O. bits to 0011 (three).

Note that you *cannot* convert a string of numeric characters to their equivalent binary representation by simply stripping the H.O. nibble from each digit in the string. Converting 123 (31h 32h 33h) in this fashion yields three bytes: 010203h, not the correct value which is 7Bh. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

Bit seven in standard ASCII is always zero. This means that the ASCII character set consumes only half of the possible character codes in an eight bit byte. IBM uses the remaining 128 character codes for various special characters including international characters (those with accents, etc.), math symbols, and line drawing characters. Note that these extra characters are a non-standard extension to the ASCII character set. Of course, the name IBM has considerable clout, so almost all modern personal computers based on the 80x86 with a video display support the extended IBM / ASCII character set. Most printers support IBM’s character set as well.

Should you need to exchange data with other machines which are not PC-compatible, you have only two alternatives: stick to standard ASCII or ensure that the target machine supports the extended IBM-PC character set. Some machines, like the Apple Macintosh, do not provide native support for the extended IBM-PC character set; however you may obtain a PC font which lets you display the extended character set. Other machines (e.g., Amiga and Atari ST) have similar capabilities. However, the 128 characters in the standard ASCII character set are the only ones you should count on transferring from system to system.

Despite the fact that it is a "standard", simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it's true that an "A" on one machine is most likely an "A" on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported – backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. End of line is a particularly troublesome example. MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Apple Macintosh, Apple II, and many other systems mark the end of line by a single CR character. UNIX systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all your files on these systems, you will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Since you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., "0", "A", "a", etc.).

1.12 Summary

Most modern computer systems use the binary numbering system to represent values. Since binary values are somewhat unwieldy, we'll often use the hexadecimal representation for those values. This is because it is very easy to convert between hexadecimal and binary, unlike the conversion between the more familiar decimal and binary systems. A single hexadecimal digit consumes four binary digits (bits), and we call a group of four bits a nibble. See:

- "The Binary Numbering System" on page 12
- "Binary Formats" on page 13
- "The Hexadecimal Numbering System" on page 17

The 80x86 works best with groups of bits which are eight, 16, or 32 bits long. We call objects of these sizes bytes, words, and double words, respectively. With a byte, we can represent any one of 256 unique values. With a word we can represent one of 65,536 different values. With a double word we can represent over four billion different values. Often we simply represent integer values (signed or unsigned) with bytes, words, and double words; however we'll often represent other quantities as well. See:

- "Data Organization" on page 13
- "Bytes" on page 14
- "Words" on page 15
- "Double Words" on page 16

In order to talk about specific bits within a nibble, byte, word, double word, or other structure, we'll number the bits starting at zero (for the least significant bit) on up to *n*-1

(where *n* is the number of bits in the object). We'll also number nibbles, bytes, and words in large structures in a similar fashion. See:

- "Binary Formats" on page 13

There are many operations we can perform on binary values including normal arithmetic (+, -, *, and /) and the logical operations (AND, OR, XOR, NOT, Shift Left, Shift Right, Rotate Left, and Rotate Right). Logical AND, OR, XOR, and NOT are typically defined for single bit operations. We can extend these to *n* bits by performing bitwise operations. The shifts and rotates are always defined for a fixed length string of bits. See:

- "Arithmetic Operations on Binary and Hexadecimal Numbers" on page 19
- "Logical Operations on Bits" on page 20
- "Logical Operations on Binary Numbers and Bit Strings" on page 22
- "Shifts and Rotates" on page 26

There are two types of integer values which we can represent with binary strings on the 80x86: unsigned integers and signed integers. The 80x86 represents unsigned integers using the standard binary format. It represents signed integers using the two's complement format. While unsigned integers may be of arbitrary length, it only makes sense to talk about fixed length signed binary values. See:

- "Signed and Unsigned Numbers" on page 23
- "Sign and Zero Extension" on page 25

Often it may not be particularly practical to store data in groups of eight, 16, or 32 bits. To conserve space you may want to pack various pieces of data into the same byte, word, or double word. This reduces storage requirements at the expense of having to perform extra operations to pack and unpack the data. See:

- "Bit Fields and Packed Data" on page 28

Character data is probably the most common data type encountered besides integer values. The IBM PC and compatibles use a variant of the ASCII character set – the extended IBM/ASCII character set. The first 128 of these characters are the standard ASCII characters, 128 are special characters created by IBM for international languages, mathematics, and line drawing. Since the use of the ASCII character set is so common in modern programs, familiarity with this character set is essential. See:

- "The ASCII Character Set" on page 28

1.13 Laboratory Exercises

Accompanying this text is a significant amount of software. This software is divided into four basic categories: source code for examples appearing throughout this text, the UCR Standard Library for 80x86 assembly language programmers, sample code you modify for various laboratory exercises, and application software to support various laboratory exercises. This software has been written using assembly language, C++, Flex/Bison, and Delphi (object Pascal). Most of the application programs include source code as well as executable code.

Much of the software accompanying this text runs under Windows 3.1, Windows 95, or Windows NT. Some software, however, directly manipulates the hardware and will only run under DOS or a DOS box in Windows 3.1. This text assumes that you are familiar with the DOS and Windows operating systems; if you are unfamiliar with DOS or Windows operation, you should refer to an appropriate text on those systems for additional details.

1.13.1 Installing the Software

The software accompanying this text is generally supplied on CD-ROM[9]. You can use most of it as-is directly off the CD-ROM. However, for speed and convenience you will probably want to install the software on a hard disk[10]. To do this, you will need to create two subdirectories in the root directory on your hard drive: ARTOFASM and STDLIB. The ARTOFASM directory will contain the files specific to this text book, the STDLIB directory will contain the files associated with the UCR Standard Library for 80x86 assembly language programmers. Once you create these two subdirectories, copy all the files and subdirectories from the corresponding directories on the CD to your hard disk. From DOS (or a DOS window), you can use the following XCOPY commands to accomplish this:

```
xcopy r:\artofasm\*.*   c:\artofasm   /s
xcopy r:\stdlib\*.*   c:\stdlib   /s
```

These commands assume that your CD-ROM is drive R: and you are installing the software on the C: hard disk. They also assume that you have created the ARTOFASM and STDLIB subdirectories prior to executing the XCOPY commands.

To use the Standard Library in programming projects, you will need to add or modify two lines in your AUTOEXEC.BAT file. If similar lines are not already present, add the following two lines to your AUTOEXEC.BAT file:

```
set lib=c:\stdlib\lib
set include=c:\stdlib\include
```

These commands tell MASM (the Microsoft Macro Assembler) where it can find the library and include files for the UCR Standard Library. Without these lines, MASM will report an error anytime you use the standard library routines in your programs.

If there are already a “set include = ...” and “set lib=...” lines in your AUTOEXEC.BAT file, you should not replace them with the lines above. Instead, you should append the string “;c:\stdlib\lib” to the end of the existing “set lib=...” statement and “;c:\stdlib\include” to the end of the existing “set include=...” statement. Several languages (like C++) also use these “set” statements; if you arbitrarily replace them with the statements above, your assembly language programs will work fine, but any attempt to compile a C++ (or other language) program may fail.

9. It is also available via anonymous ftp, although there are many files associated with this text.

10. If you are using this software in a laboratory at school, your instructor has probably installed this software on the machines in the laboratory. As a general rule, you should never install software on machines in the laboratory. Check with your laboratory instruction before installing this software on machines in the laboratory.

If you forget to put these lines in your AUTOEXEC.BAT file, you can temporarily (until the next time you boot the system) issue these commands by simply typing them at the DOS command line prompt. By typing "set" by itself on the command line prompt, you can see if these set commands are currently active.

If you do not have a CD-ROM player, you can obtain the software associated with this textbook via anonymous ftp from cs.ucr.edu. Check in the "/pub/pc/ibmpc" subdirectory. The files on the ftp server will be compressed. A "README" file will describe how to decompress the data.

The STDLIB directory you've created holds the source and library files for the UCR Standard Library for 80x86 Assembly Language Programmers. This is a core set of assembly language subroutines you can call that mimic many of the routines in the C standard library. These routines greatly simplify writing programs in assembly language. Furthermore, they are public domain so you can use them in any programs you write without fear of licensing restrictions.

The ARTOFASM directory contains files specific to this text. Within the ARTOFASM directory you will see a sequence of subdirectories named ch1, ch2, ch3, etc. These subdirectories contain the files associated with Chapter One, Chapter Two, and so on. Within some of these subdirectories, you will find two subdirectories named "DOS" and "WINDOWS". If these subdirectories are present, they separate those files that must run under MS-Windows from those that run under DOS. *Many of the DOS programs require a "real-mode" environment and will not run in a DOS box window in Windows 95 or Windows NT.* You will need to run this software directory from MS-DOS. The Windows applications require a color monitor.

There is often a third subdirectory present in each chapter directory: SOURCES. This subdirectory contains the source listings (where appropriate or feasible) to the software for that chapter. Most of the software for this text is written in assembly language using MASM 6.x, generic C++, Turbo Pascal, or Borland Delphi (visual object Pascal). If you are interested in seeing how the software operates, you can look in this subdirectory.

This text assumes you already know how to run programs from MS-DOS and Windows and you are familiar with common DOS and Windows terminology. It also assumes you know some simple MS-DOS commands like DIR, COPY, DEL, RENAME, and so on. If you are new to Windows and DOS, you should pick up an appropriate reference manual on these operating systems.

The files for Chapter One's laboratory exercises appear in the ARTOFASM\CH1 subdirectory. These are all Windows programs, so you will need to be running Windows 3.1, Windows 95, Windows NT, or some later (and compatible) version of Windows to run these programs.

1.13.2 Data Conversion Exercises

In this exercise you will be using the "convert.exe" program found in the ARTOFASM\CH1 subdirectory. This program displays and converts 16-bit integers using signed decimal, unsigned decimal, hexadecimal, and binary notation.

When you run this program it opens a window with four *edit boxes*. (one for each data type). Changing a value in one of the edit boxes immediately updates the values in the other boxes so they all display their corresponding representations for the new value. If you make a mistake on data entry, the program beeps and turns the edit box red until you correct the mistake. Note that you can use the mouse, cursor control keys, and the editing keys (e.g., DEL and Backspace) to change individual values in the edit boxes.

For this exercise and your laboratory report, you should explore the relationship between various binary, hexadecimal, unsigned decimal, and signed decimal values. For example, you should enter the unsigned decimal values 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, and 32768 and comment on the values that appear in the in the other text boxes.

The primary purpose of this exercise is to familiarize yourself with the decimal equivalents of some common binary and hexadecimal values. In your lab report, for example, you should explain what is special about the binary (and hexadecimal) equivalents of the decimal numbers above.

Another set of experiments to try is to choose various binary numbers that have exactly two bits set, e.g., 11, 110, 1100, 1 1000, 11 0000, etc. Be sure to comment on the decimal and hexadecimal results these inputs produce.

Try entering several binary numbers where the L.O. eight bits are all zero. Comment on the results in your lab report. Try the same experiment with hexadecimal numbers using zeros for the L.O. digit or the two L.O. digits.

You should also experiment with negative numbers in the signed decimal text entry box; try using values like -1, -2, -3, -256, -1024, etc. Explain the results you obtain using your knowledge of the two's complement numbering system.

Try entering even and odd numbers in unsigned decimal. Discover and describe the difference between even and odd numbers in their binary representation. Try entering multiples of other values (e.g., for three: 3, 6, 9, 12, 15, 18, 21, ...) and see if you can detect a pattern in the binary results.

Verify the hexadecimal <-> binary conversion this chapter describes. In particular, enter the same hexadecimal digit in each of the four positions of a 16-bit value and comment on the position of the corresponding bits in the binary representation. Try several entering binary values like 1111, 11110, 111100, 1111000, and 11110000. Explain the results you get and describe why you should always extend binary values so their length is an even multiple of four before converting them.

In your lab report, list the experiments above plus several you devise yourself. Explain the results you expect and include the actual results that the convert.exe program produces. Explain any insights you have while using the convert.exe program.

1.13.3 Logical Operations Exercises

The logical.exe program is a simple calculator that computes various logical functions. It allows you to enter binary or hexadecimal values and then it computes the result of some logical operation on the inputs. The calculator supports the dyadic logical AND, OR, and XOR. It also supports the monadic NOT, NEG (two's complement), SHL (shift left), SHR (shift right), ROL (rotate left), and ROR (rotate right).

When you run the logical.exe program it displays a set of buttons on the left hand side of the window. These buttons let you select the calculation. For example, pressing the AND button instructs the calculator to compute the logical AND operation between the two input values. If you select a monadic (unary) operation like NOT, SHL, etc., then you may only enter a single value; for the dyadic operations, both sets of text entry boxes will be active.

The logical.exe program lets you enter values in binary or hexadecimal. Note that this program automatically converts any changes in the binary text entry window to hexadecimal and updates the value in the hex entry edit box. Likewise, any changes in the hexadecimal text entry box are immediately reflected in the binary text box. If you enter an illegal value in a text entry box, the logical.exe program will turn the box red until you correct the problem.

For this laboratory exercise, you should explore each of the bitwise logical operations. Create several experiments by carefully choosing some values, manually compute the result you expect, and then run the experiment using the logical.exe program to verify your results. You should especially experiment with the masking capabilities of the logical AND, OR, and XOR operations. Try logically ANDing, ORing, and XORing different values with values like 000F, 00FF, 00F0, 0FFF, FF00, etc. Report the results and comment on them in your laboratory report.

Some experiments you might want to try, in addition to those you devise yourself, include the following:

- Devise a mask to convert ASCII values '0'..'9' to their binary integer counterparts using the logical AND operation. Try entering the ASCII codes of each of these digits when using this mask. Describe your results. What happens if you enter non-digit ASCII codes?
- Devise a mask to convert integer values in the range 0..9 to their corresponding ASCII codes using the logical OR operation. Enter each of the binary values in the range 0..9 and describe your results. What happens if you enter values outside the range 0..9? In particular, what happens if you enter values outside the range 0h..0fh?
- Devise a mask to determine whether a 16-bit integer value is positive or negative using the logical AND operation. The result should be zero if the number is positive (or zero) and it should be non-zero if the number is negative. Enter several positive and negative values to test your mask. Explain how you could use the AND operation to test *any* single bit to determine if it is zero or one.
- Devise a mask to use with the logical XOR operation that will produce the same result on the second operand as applying the logical NOT operator to that second operand.
- Verify that the SHL and SHR operators correspond to an integer multiplication by two and an integer division by two, respectively. What happens if you shift data out of the H.O. or L.O. bits? What does this correspond to in terms of integer multiplication and division?
- Apply the ROL operation to a set of positive and negative numbers. Based on your observations in Section 1.13.3, what can you say will about the result when you rotate left a negative number or a positive number?
- Apply the NEG and NOT operators to a value. Discuss the similarity and the difference in their results. Describe this difference based on your knowledge of the two's complement numbering system.

1.13.4 Sign and Zero Extension Exercises

The signext.exe program accepts eight-bit binary or hexadecimal values then sign and zero extends them to 16 bits. Like the logical.exe program, this program lets you enter a value in either binary or hexadecimal and immediate zero and sign extends that value.

For your laboratory report, provide several eight-bit input values and describe the results you expect. Run these values through the signext.exe program and verify the results. For each experiment you run, be sure to list all the results in your lab report. Be sure to try values like 0, 7fh, 80h, and 0ffh.

While running these experiments, discover which hexadecimal digits appearing in the H.O. nibble produce negative 16-bit numbers and which produce positive 16-bit values. Document this set in your lab report.

Enter sets of values like (1,10), (2,20), (3,30), ..., (7,70), (8,80), (9,90), (A,A0), ..., (F,F0). Explain the results you get in your lab report. Why does "F" sign extend with zeros while "F0" sign extends with ones?

Explain in your lab report how one would sign or zero extend 16 bit values to 32 bit values. Explain why zero extension or sign extension is useful.

1.13.5 Packed Data Exercises

The packdata.exe program uses the Date data type appearing in this chapter (see “Bit Fields and Packed Data” on page 28). It lets you input a date value in binary or decimal and it packs that date into a single 16-bit value.

When you run this program, it will give you a window with six data entry boxes: three to enter the date in decimal form (month, day, year) and three text entry boxes that let you enter the date in binary form. The month value should be in the range 1..12, the day value should be in the range 1..31, and the year value should be in the range 0..99. If you enter a value outside this range (or some other illegal value), then the packdata.exe program will turn the data entry box red until you correct the problem.

Choose several dates for your experiments and convert these dates to the 16-bit packed binary form by hand (if you have trouble with the decimal to binary conversion, use the conversion program from the first set of exercises in this laboratory). Then run these dates through the packdata.exe program to verify your answer. Be sure to include all program output in your lab report.

At a bare minimum, you should include the following dates in your experiments:

2/4/68, 1/1/80, 8/16/64, 7/20/60, 11/2/72, 12/25/99, Today’s Date, a birthday (not necessarily yours), the due date on your lab report.

1.14 Questions

1) Convert the following decimal values to binary:

a) 128	b) 4096	c) 256	d) 65536	e) 254
f) 9	g) 1024	h) 15	i) 344	j) 998
k) 255	l) 512	m) 1023	n) 2048	o) 4095
p) 8192	q) 16,384	r) 32,768	s) 6,334	t) 12,334
u) 23,465	v) 5,643	w) 464	x) 67	y) 888

2) Convert the following binary values to decimal:

a) 1001 1001	b) 1001 1101	c) 1100 0011	d) 0000 1001	e)1111 1111
f) 0000 1111	g) 0111 1111	h) 1010 0101	i) 0100 0101	j) 0101 1010
k) 1111 0000	l) 1011 1101	m) 1100 0010	n) 0111 1110	o) 1110 1111
p) 0001 1000	q) 1001 111 1	r) 0100 0010	s) 1101 1100	t) 1111 0001
u) 0110 1001	v) 0101 1011	w) 1011 1001	x) 1110 0110	y) 1001 0111

3) Convert the binary values in problem 2 to hexadecimal.

4) Convert the following hexadecimal values to binary:

a) 0ABCD	b) 1024	c) 0DEAD	d) 0ADD	e) 0BEEF
f) 8	g) 05AAF	h) 0FFFF	i) 0ACDB	j) 0CDBA
k) 0FEBA	l) 35	m) 0BA	n) 0ABA	o) 0BAD
p) 0DAB	q) 4321	r) 334	s) 45	t) 0E65
u) 0BEAD	v) 0ABE	w) 0DEAF	x) 0DAD	y) 9876

Perform the following hex computations (leave the result in hex):

5) 1234 +9876

6) 0FFF - 0F34

7) 100 - 1

8) 0FFE - 1

9) What is the importance of a nibble?

10) How many hexadecimal digits in:

a) a byte b) a word c) a double word

11) How many bits in a:

a) nibble b) byte c) word d) double word

12) Which bit (number) is the H.O. bit in a:

a) nibble b) byte c) word d) double word

13) What character do we use as a suffix for hexadecimal numbers? Binary numbers? Decimal numbers?

14) Assuming a 16-bit two's complement format, determine which of the values in question 4 are positive and which are negative.

15) Sign extend all of the values in question two to sixteen bits. Provide your answer in hex.

16) Perform the bitwise AND operation on the following pairs of hexadecimal values. Present your answer in hex. (Hint: convert hex values to binary, do the operation, then convert back to hex).

a) 0FF00, 0FF0 b) 0F00F, 1234 c) 4321, 1234 d) 2341, 3241 e) 0FFFF, 0EDCB

f) 1111, 5789 g) 0FABA, 4322 h) 5523, 0F572 i) 2355, 7466 j) 4765, 6543

k) 0ABCD, 0EFDCl) 0DDDD, 1234m) 0CCCC, 0ABCDn) 0BBBB, 1234o) 0AAAA, 1234

p) 0EEEE, 1248 q) 8888, 1248 r) 8086, 124F s) 8086, 0CFA7 t) 8765, 3456

u) 7089, 0FEDC v) 2435, 0BCDE w) 6355, 0EFDC x) 0CBA, 6884 y) 0AC7, 365

17) Perform the logical OR operation on the above pairs of numbers.

18) Perform the logical XOR operation on the above pairs of numbers.

19) Perform the logical NOT operation on all the values in question four. Assume all values are 16 bits.

20) Perform the two's complement operation on all the values in question four. Assume 16 bit values.

21) Sign extend the following hexadecimal values from eight to sixteen bits. Present your answer in hex.

a) FF	b) 82	c) 12	d) 56	e) 98
f) BF	g) 0F	h) 78	i) 7F	j) F7
k) 0E	l) AE	m) 45	n) 93	o) C0
p) 8F	q) DA	r) 1D	s) 0D	t) DE
u) 54	v) 45	w) F0	x) AD	y) DD

22) Sign contract the following values from sixteen bits to eight bits. If you cannot perform the operation, explain why.

a) FF00	b) FF12	c) FFF0	d) 12	e) 80
f) FFFF	g) FF88	h) FF7F	i) 7F	j) 2
k) 8080	l) 80FF	m) FF80	n) FF	o) 8
p) F	q) 1	r) 834	s) 34	t) 23
u) 67	v) 89	w) 98	x) FF98	y) F98

23) Sign extend the 16-bit values in question 22 to 32 bits.

24) Assuming the values in question 22 are 16-bit values, perform the left shift operation on them.

25) Assuming the values in question 22 are 16-bit values, perform the right shift operation on them.

26) Assuming the values in question 22 are 16-bit values, perform the rotate left operation on them.

27) Assuming the values in question 22 are 16-bit values, perform the rotate right operation on them.

28) Convert the following dates to the packed format described in this chapter (see "Bit Fields and Packed Data" on page 28). Present your values as a 16-bit hex number.

a) 1/1/92 b) 2/4/56 c) 6/19/60 d) 6/16/86 e) 1/1/99

29) Describe how to use the shift and logical operations to *extract* the day field from the packed date record in question 28. That is, wind up with a 16-bit integer value in the range 0..31.

30) Suppose you have a value in the range 0..9. Explain how you could convert it to an ASCII character using the basic logical operations.

31) The following C++ function locates the first set bit in the BitMap parameter starting at bit position start and working up to the H.O. bit. If no such bit exists, it returns -1. Explain, in detail, how this function works.

```
int FindFirstSet(unsigned BitMap, unsigned start)
{
	unsigned Mask = (1 << start);

	while (Mask)
	{
		if (BitMap & Mask) return start;
		++start;
		Mask <<= 1;
	}
	return -1;
}
```

32) The C++ programming language does not specify how many bits there are in an unsigned integer. Explain why the code above will work regardless of the number of bits in an unsigned integer.

33) The following C++ function is the complement to the function in the questions above. It locates the first zero bit in the BitMap parameter. Explain, in detail, how it accomplishes this.

```
int FindFirstClr(unsigned BitMap, unsigned start)
{
	return FindFirstSet(~BitMap, start);
}
```

34) The following two functions set or clear (respectively) a particular bit and return the new result. Explain, in detail, how these functions operate.

```
unsigned SetBit(unsigned BitMap, unsigned position)
{
	return BitMap | (1 << position);
}

unsigned ClrBit(unsigned BitMap, unsigned position)
{
	return BitMap & ~(1 << position);
}
```

35) In code appearing in the questions above, explain what happens if the start and position parameters contain a value greater than or equal to the number of bits in an unsigned integer.

1.15 Programming Projects

The following programming projects assume you are using C, C++, Turbo Pascal, Borland Pascal, Delphi, or some other programming language that supports bitwise logical operations. Note that C and C++ use the "&", "|", and "^" operators for logical AND, OR, and XOR, respectively. The Borland Pascal products let you use the "and", "or", and "xor" operators on integers to perform bitwise logical operations. The following projects all expect you to use these logical operators. There are other solutions to these problems that do not involve the use of logical operations, **do not employ such a solution**. The purpose of these exercises is to introduce you to the logical operations available in high level languages. **Be sure to check with your instructor to determine which language you are to use.**

The following descriptions typically describe functions you are to write. However, you will need to write a main program to call and test each of the functions you write as part of the assignment.

1) Write to functions, *toupper* and *tolower*, that take a single character as their parameter and convert this character to upper case (if it was lowercase) or to lowercase (if it was uppercase) respectively. Use the logical operations to do the conversion. Pascal users may need to use the chr() and ord() functions to successfully complete this assignment.

2) Write a function "CharToInt" that you pass a string of characters and it returns the corresponding integer value. *Do not use a built-in library routine like atoi (C) or strtoint (Pascal) to do this conversion*. You are to process each character passed in the input string, convert it from a character to an integer using the logical operations, and accumulate the result until you reach the end of the string. An easy algorithm for this task is to multiply the accumulated result by 10 and then add in the next digit. Repeat this until you reach the end of the string. Pascal users will probably need to use the ord() function in this assignment.

3) Write a ToDate function that accepts three parameters, a month, day, and year value. This function should return the 16-bit packed date value using the format given in this chapter (see "Bit Fields and Packed Data" on page 28). Write three corresponding functions ExtractMonth, ExtractDay, and ExtractYear that expect a 16-bit date value and return the corresponding month, day, or year value. The ToDate function should automatically convert dates in the range 1900-1999 to the range 0..99.

4) Write a "CntBits" function that counts the number of one bits in a 16-bit integer value. *Do not use any built-in functions in your language's library to count these bits for you.*

5) Write a "TestBit" function. This function requires two 16-bit integer parameters. The first parameter is a 16-bit value to test; the second parameter is a value in the range 0..15 describing which bit to test. The function should return true if the corresponding bit contains a one, the function should return false if that bit position contains a zero. The function should always return false if the second parameter holds a value outside the range 0..15.

6) Pascal and C/C++ provide shift left and shift right operators (SHL/SHR in Pascal, "<<" and ">>" in C/C++). However, they do not provide rotate right and rotate left operators. Write a pair of functions, ROL and ROR, that perform the rotate tasks. Hint: use the function from exercise five to test the H.O. bit. Then use the corresponding shift operation and the logical OR operation to perform the rotate.

Boolean Algebra Chapter Two

Logic circuits are the basis for modern digital computer systems. To appreciate how computer systems operate you will need to understand digital logic and boolean algebra.

This Chapter provides only a basic introduction to boolean algebra. This subject alone is often the subject of an entire textbook. This Chapter will concentrate on those subject that support other chapters in this text.

2.0 Chapter Overview

Boolean logic forms the basis for computation in modern binary computer systems. You can represent any algorithm, or any electronic computer circuit, using a system of boolean equations. This chapter provides a brief introduction to boolean algebra, truth tables, canonical representation, of boolean functions, boolean function simplification, logic design, combinatorial and sequential circuits, and hardware/software equivalence.

The material is especially important to those who want to design electronic circuits or write software that controls electronic circuits. Even if you never plan to design hardware or write software than controls hardware, the introduction to boolean algebra this chapter provides is still important since you can use such knowledge to optimize certain complex conditional expressions within IF, WHILE, and other conditional statements.

The section on minimizing (optimizing) logic functions uses *Veitch Diagrams* or *Karnaugh Maps*. The optimizing techniques this chapter uses reduce the number of *terms* in a boolean function. You should realize that many people consider this optimization technique obsolete because reducing the number of terms in an equation is not as important as it once was. This chapter uses the mapping method as an example of boolean function optimization, not as a technique one would regularly employ. If you are interested in circuit design and optimization, you will need to consult a text on logic design for better techniques.

Although this chapter is mainly hardware-oriented, keep in mind that many concepts in this text will use boolean equations (logic functions). Likewise, some programming exercises later in this text will assume this knowledge. Therefore, you should be able to deal with boolean functions before proceeding in this text.

2.1 Boolean Algebra

Boolean algebra is a deductive mathematical system closed over the values zero and one (false and true). A *binary operator* "°" defined over this set of values accepts a pair of boolean inputs and produces a single boolean value. For example, the boolean AND operator accepts two boolean inputs and produces a single boolean output (the logical AND of the two inputs).

For any given algebra system, there are some initial assumptions, or *postulates*, that the system follows. You can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems often employ the following postulates:

- *Closure*. The boolean system is *closed* with respect to a binary operator if for every pair of boolean values, it produces a boolean result. For example, logical AND is closed in the boolean system because it accepts only boolean operands and produces only boolean results.
- *Commutativity*. A binary operator "°" is said to be commutative if A°B = B°A for all possible boolean values A and B.

- *Associativity*. A binary operator "°" is said to be associative if

  $$(A \circ B) \circ C = A \circ (B \circ C)$$

  for all boolean values A, B, and C.

- *Distribution*. Two binary operators "°" and "%" are distributive if

  $$A \circ (B \,\%\, C) = (A \circ B) \,\%\, (A \circ C)$$

  for all boolean values A, B, and C.

- *Identity*. A boolean value I is said to be the *identity element* with respect to some binary operator "°" if $A \circ I = A$.

- *Inverse*. A boolean value I is said to be the *inverse element* with respect to some binary operator "°" if $A \circ I = B$ and $B \neq A$ (i.e., B is the opposite value of A in a boolean system).

For our purposes, we will base boolean algebra on the following set of operators and values:

The two possible values in the boolean system are zero and one. Often we will call these values false and true (respectively).

The symbol "•" represents the logical AND operation; e.g., A • B is the result of logically ANDing the boolean values A and B. When using single letter variable names, this text will drop the "•" symbol; Therefore, AB also represents the logical AND of the variables A and B (we will also call this the *product* of A and B).

The symbol "+" represents the logical OR operation; e.g., A + B is the result of logically ORing the boolean values A and B. (We will also call this the *sum* of A and B.)

Logical *complement*, *negation*, or *not*, is a unary operator. This text will use the (') symbol to denote logical negation. For example, A' denotes the logical NOT of A.

If several different operators appear in a single boolean expression, the result of the expression depends on the *precedence* of the operators. We'll use the following precedences (from highest to lowest) for the boolean operators: parenthesis, logical NOT, logical AND, then logical OR. The logical AND and OR operators are *left associative*. If two operators with the same precedence are adjacent, you must evaluate them from left to right. The logical NOT operation is right associative, although it would produce the same result using left or right associativity since it is a unary operator.

We will also use the following set of postulates:

P1 Boolean algebra is closed under the AND, OR, and NOT operations.

P2 The identity element with respect to • is one and + is zero. There is no identity element with respect to logical NOT.

P3 The • and + operators are commutative.

P4 • and + are distributive with respect to one another. That is, A • (B + C) = (A • B) + (A • C) and A + (B • C) = (A + B) • (A + C).

P5 For every value A there exists a value A' such that A•A' = 0 and A+A' = 1. This value is the logical complement (or NOT) of A.

P6 • and + are both associative. That is, (A•B)•C = A•(B•C) and (A+B)+C = A+(B+C).

You can prove all other theorems in boolean algebra using these postulates. This text will not go into the formal proofs of these theorems, however, it is a good idea to familiarize yourself with some important theorems in boolean algebra. A sampling include:

Th1: A + A = A

Th2: A • A = A

Th3: A + 0 = A

Th4: A • 1 = A

Th5: A • 0 = 0

Th6: A + 1 = 1

Th7: (A + B)' = A' • B'

Th8: (A • B)' = A' + B'

Th9: A + A•B = A

Th10: A •(A + B) = A

Th11: A + A'B = A+B

Th12: A' • (A + B') = A'B'

Th13: AB + AB' = A

Th14: (A'+B') • (A' + B) = A'

Th15: A + A' = 1

Th16: A • A' = 0

Theorems seven and eight above are known as *DeMorgan's Theorems* after the mathematician who discovered them.

The theorems above appear in pairs. Each pair (e.g., Th1 & Th2, Th3 & Th4, etc.) form a *dual*. An important principle in the boolean algebra system is that of *duality*. Any valid expression you can create using the postulates and theorems of boolean algebra remains valid if you interchange the operators and constants appearing in the expression. Specifically, if you exchange the • and + operators and swap the 0 and 1 values in an expression, you will wind up with an expression that obeys all the rules of boolean algebra. *This does not mean the dual expression computes the same values*, it only means that both expressions are legal in the boolean algebra system. Therefore, this is an easy way to generate a second theorem for any fact you prove in the boolean algebra system.

Although we will not be proving any theorems for the sake of boolean algebra in this text, we will use these theorems to show that two boolean equations are identical. This is an important operation when attempting to produce *canonical representations* of a boolean expression or when simplifying a boolean expression.

2.2 Boolean Functions and Truth Tables

A boolean *expression* is a sequence of zeros, ones, and *literals* separated by boolean operators. A literal is a primed (negated) or unprimed variable name. For our purposes, all variable names will be a single alphabetic character. A boolean function is a specific boolean expression; we will generally give boolean functions the name "F" with a possible subscript. For example, consider the following boolean:

$$F_0 = AB+C$$

This function computes the logical AND of A and B and then logically ORs this result with C. If A=1, B=0, and C=1, then F_0 returns the value one (1•0 + 1 = 1).

Another way to represent a boolean function is via a *truth table*. The previous chapter used truth tables to represent the AND and OR functions. Those truth tables took the forms:

Table 6: AND Truth Table

AND	0	1
0	0	0
1	0	1

Table 7: OR Truth Table

OR	0	1
0	0	1
1	1	1

For binary operators and two input variables, this form of a truth table is very natural and convenient. However, reconsider the boolean function F_0 above. That function has *three* input variables, not two. Therefore, one cannot use the truth table format given above. Fortunately, it is still very easy to construct truth tables for three or more variables. The following example shows one way to do this for functions of three or four variables:

Table 8: Truth Table for a Function with Three Variables

F = AB + C		BA			
		00	01	10	11
C	0	0	0	0	1
	1	1	1	1	1

Table 9: Truth Table for a Function with Four Variables

F = AB + CD		BA			
		00	01	10	11
DC	00	0	0	0	1
	01	0	0	0	1
	10	0	0	0	1
	11	1	1	1	1

In the truth tables above, the four columns represent the four possible combinations of zeros and ones for A & B (B is the H.O. or leftmost bit, A is the L.O. or rightmost bit). Likewise the four rows in the second truth table above represent the four possible combinations of zeros and ones for the C and D variables. As before, D is the H.O. bit and C is the L.O. bit.

Table 10 shows another way to represent truth tables. This form has two advantages over the forms above – it is easier to fill in the table and it provides a compact representation for two or more functions.

Note that the truth table above provides the values for three separate functions of three variables.

Although you can create an infinite variety of boolean functions, they are not all unique. For example, F=A and F=AA are two different functions. By theorem two, however, it is easy to show that these two functions are equivalent, that is, they produce exactly the same outputs for all input combinations. If you fix the number of input variables, there are a finite number of unique boolean functions possible. For example, there are only 16 unique boolean functions with two inputs and there are only 256 possible boolean functions of three input variables. Given *n* input variables, there are $2^{**}(2^n)$ (two raised to the two raised to the nth power) unique boolean functions of those *n* input values. For two input variables, $2^{\wedge}(2^2) = 2^4$ or 16 different functions. With three input vari-

Table 10: Another Format for Truth Tables

C	B	A	F = ABC	F = AB + C	F = A+BC
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

ables there are $2^{**}(2^3) = 2^8$ or 256 possible functions. Four input variables create $2^{**}(2^4)$ or 2^{16}, or 65,536 different unique boolean functions.

When dealing with only 16 boolean functions, it's easy enough to name each function. The following table lists the 16 possible boolean functions of two input variables along with some common names for those functions:

Table 11: The 16 Possible Boolean Functions of Two Variables

Function #	Description
0	Zero or Clear. Always returns zero regardless of A and B input values.
1	Logical NOR (NOT (A OR B)) = (A+B)'
2	Inhibition = BA' (B, not A). Also equivalent to B>A or A < B.
3	NOT A. Ignores B and returns A'.
4	Inhibition = AB' (A, not B). Also equivalent to A>B or B<A.
5	NOT B. Returns B' and ignores A
6	Exclusive-or (XOR) = A $\oplus$ B. Also equivalent to A≠B.
7	Logical NAND (NOT (A AND B)) = (A•B)'
8	Logical AND = A•B. Returns A AND B.
9	Equivalence = (A = B). Also known as exclusive-NOR (not exclusive-or).
10	Copy B. Returns the value of B and ignores A's value.
11	Implication, B implies A = A + B'. (if B then A). Also equivalent to B >= A.
12	Copy A. Returns the value of A and ignores B's value.
13	Implication, A implies B = B + A' (if A then B). Also equivalent to A >= B.
14	Logical OR = A+B. Returns A OR B.
15	One or Set. Always returns one regardless of A and B input values.

Beyond two input variables there are too many functions to provide specific names. Therefore, we will refer to the function's number rather than the function's name. For example, F_8 denotes the logical AND of A and B for a two-input function and F_{14} is the logical OR operation. Of course, the only problem is to determine a function's number. For

example, given the function of three variables F=AB+C, what is the corresponding function number? This number is easy to compute by looking at the truth table for the function (see Table 14 on page 50). If we treat the values for A, B, and C as bits in a binary number with C being the H.O. bit and A being the L.O. bit, they produce the binary numbers in the range zero through seven. Associated with each of these binary strings is a zero or one function result. If we construct a binary value by placing the function result in the bit position specified by A, B, and C, the resulting binary number is that function's number. Consider the truth table for F=AB+C:

CBA:	7	6	5	4	3	2	1	0
F=AB+C:	1	1	1	1	1	0	0	0

If we treat the function values for F as a binary number, this produces the value $F8_{16}$ or 248_{10}. We will usually denote function numbers in decimal.

This also provides the insight into why there are $2^{**}2^n$ different functions of *n* variables: if you have *n* input variables, there are 2^n bits in function's number. If you have *m* bits, there are 2^m different values. Therefore, for *n* input variables there are $m=2^n$ possible bits and 2^m or $2^{**}2^n$ possible functions.

2.3 Algebraic Manipulation of Boolean Expressions

You can transform one boolean expression into an equivalent expression by applying the postulates the theorems of boolean algebra. This is important if you want to convert a given expression to a *canonical form* (a standardized form) or if you want to minimize the number of literals (primed or unprimed variables) or terms in an expression. Minimizing terms and expressions can be important because electrical circuits often consist of individual components that implement each term or literal for a given expression. Minimizing the expression allows the designer to use fewer electrical components and, therefore, can reduce the cost of the system.

Unfortunately, there are no fixed rules you can apply to optimize a given expression. Much like constructing mathematical proofs, an individual's ability to easily do these transformations is usually a function of experience. Nevertheless, a few examples can show the possibilities:

```
ab + ab' + a'b          =  a(b+b') + a'b              By P4
                        =  a•1 + a'b                  By P5
                        =  a + a'b                    By Th4
                        =  a + a'b + 0                By Th3
                        =  a + a'b + aa'              By P5
                        =  a + b(a + a')              By P4
                        =  a + b•1                    By P5
                        =  a + b                      By Th4


(a'b + a'b' + b')'      =  ( a'(b+b') + b')'          By P4
                        =  (a' + b')'                 By P5
                        =  ( (ab)' )'                 By Th8
                        =  ab                         By definition of not


b(a+c) + ab' + bc' + c  =  ba + bc + ab' + bc' + c    By P4
                        =  a(b+b') + b(c + c') + c    By P4
                        =  a•1 + b•1 + c              By P5
                        =  a + b + c                  By Th4
```

Although these examples all use algebraic transformations to simplify a boolean expression, we can also use algebraic operations for other purposes. For example, the next section describes a canonical form for boolean expressions. Canonical forms are rarely optimal.

2.4 Canonical Forms

Since there are a finite number of boolean functions of *n* input variables, yet an infinite number of possible logic expressions you can construct with those *n* input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given the same inputs). To help eliminate possible confusion, logic designers generally specify a boolean function using a *canonical*, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called *sum of minterms* and the second is the *product of maxterms*. Using the duality principle, it is very easy to convert between these two.

A *term* is a variable or a product (logical AND) of several different literals. For example, if you have two variables, A and B, there are eight possible terms: A, B, A', B', A'B', A'B, AB', and AB. For three variables we have 26 different terms: A, B, C, A', B', C', A'B', A'B, AB', AB, A'C', A'C, AC', AC, B'C', B'C, BC', BC, A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. As you can see, as the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing exactly *n* literals. For example, the minterms for two variables are A'B', AB', A'B, and AB. Likewise, the minterms for three variables A, B, and C are A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. In general, there are 2^n minterms for *n* variables. The set of possible minterms is very easy to generate since they correspond to the sequence of binary numbers:

Table 12: Minterms for Three Input Variables

Binary Equivalent (CBA)	Minterm
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

We can specify *any* boolean function using a sum (logical OR) of minterms. Given F_{248}=AB+C the equivalent canonical form is ABC+A'BC+AB'C+A'B'C+ABC'. Algebraically, we can show that these two are equivalent as follows:

```
ABC+A'BC+AB'C+A'B'C+ABC' =  BC(A+A') + B'C(A+A') + ABC'
                         =  BC•1 +B'C•1 + ABC'
                         =  C(B+B') + ABC'
                         =  C + ABC'
                         =  C + AB
```

Obviously, the canonical form is not the optimal form. On the other hand, there is a big advantage to the sum of minterms canonical form: it is very easy to generate the truth table for a function from this canonical form. Furthermore, it is also very easy to generate the logic equation from the truth table.

To build the truth table from the canonical form, simply convert each minterm into a binary value by substituting a "1" for unprimed variables and a "0" for primed variables.

Then place a “1” in the corresponding position (specified by the binary minterm value) in the truth table:

1) Convert minterms to binary equivalents:

F_{248}= CBA + CBA’ + CB’A + CB’A’ + C’BA

= 111 + 110 + 101 + 100 + 011

2) Substitute a one in the truth table for each entry above

Table 13: Creating a Truth Table from Minterms, Step One

C	B	A	F = AB+C
0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Finally, put zeros in all the entries that you did not fill with ones in the first step above:

Table 14: Creating a Truth Table from Minterms, Step Two

C	B	A	F = AB+C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Going in the other direction, generating a logic function from a truth table, is almost as easy. First, locate all the entries in the truth table with a one. In the table above, these are the last five entries. The number of table entries containing ones determines the number of minterms in the canonical equation. To generate the individual minterms, substitute A, B, or C for ones and A’, B’, or C’ for zeros in the truth table above. Then compute the sum of these items. In the example above, F_{248} contains one for CBA = 111, 110, 101, 100, and 011. Therefore, F_{248} = CBA + CBA’ + CB’A + CB’A’ + C’AB. The first term, CBA, comes from the last entry in the table above. C, B, and A all contain ones so we generate the minterm CBA (or ABC, if you prefer). The second to last entry contains 110 for CBA, so we generate the minterm CBA’. Likewise, 101 produces CB’A; 100 produces CB’A’, and 011 produces C’BA. Of course, the logical OR and logical AND operations are both commutative, so we can rearrange the terms within the minterms as we please and we can rearrange the minterms within the sum as we see fit. This process works equally well for any number of

variables. Consider the function F_{53504} = ABCD + A'BCD + A'B'CD + A'B'C'D. Placing ones in the appropriate positions in the truth table generates the following:

Table 15: Creating a Truth Table with Four Variables from Minterms

D	C	B	A	F = ABCD + A'BCD + A'B'CD + A'B'C'D
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	1
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	1

The remaining elements in this truth table all contain zero.

Perhaps the easiest way to generate the canonical form of a boolean function is to first generate the truth table for that function and then build the canonical form from the truth table. We'll use this technique, for example, when converting between the two canonical forms this chapter presents. However, it is also a simple matter to generate the sum of minterms form algebraically. By using the distributive law and theorem 15 (A + A' = 1) makes this task easy. Consider F_{248} = AB + C. This function contains two terms, AB and C, but they are not minterms. Minterms contain each of the possible variables in a primed or unprimed form. We can convert the first term to a sum of minterms as follows:

```
AB    =    AB • 1                          By Th4
       =    AB • (C + C')                   By Th 15
       =    ABC + ABC'                      By distributive law
       =    CBA + C'BA                      By associative law
```

Similarly, we can convert the second term in F_{248} to a sum of minterms as follows:

```
C     =    C • 1                           By Th4
       =    C • (A + A')                    By Th15
       =    CA + CA'                        By distributive law
       =    CA•1 + CA'•1                    By Th4
       =    CA • (B + B') + CA' • (B + B')  By Th15
       =    CAB + CAB' + CA'B + CA'B'       By distributive law
       =    CBA + CBA' + CB'A + CB'A'       By associative law
```

The last step (rearranging the terms) in these two conversions is optional. To obtain the final canonical form for F_{248} we need only sum the results from these two conversions:

```
F248  =    (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A')
       =    CBA + CBA' + CB'A + CB'A' + C'BA
```

Another way to generate a canonical form is to use *products of maxterms*. A maxterm is the sum (logical OR) of all input variables, primed or unprimed. For example, consider the following logic function G of three variables:

```
G = (A+B+C) • (A'+B+C) • (A+B'+C).
```

Like the sum of minterms form, there is exactly one product of maxterms for each possible logic function. Of course, for every product of maxterms there is an equivalent sum of minterms form. In fact, the function G, above, is equivalent to

$F_{248} = CBA + CBA' + CB'A + CB'A' + C'BA = AB + C.$

Generating a truth table from the product of maxterms is no more difficult than building it from the sum of minterms. You use the duality principle to accomplish this. Remember, the duality principle says to swap AND for OR and zeros for ones (and vice versa). Therefore, to build the truth table, you would first swap primed and non-primed literals. In G above, this would yield:

$G = (A' + B' + C') \cdot (A + B' + C') \cdot (A' + B + C')$

The next step is to swap the logical OR and logical AND operators. This produces

$G = A'B'C' + AB'C' + A'BC'$

Finally, you need to swap all zeros and ones. This means that you store *zeros* into the truth table for each of the above entries and then fill in the rest of the truth table with ones. This will place a zero in entries zero, one, and two in the truth table. Filling the remaining entries with ones produces F_{248}.

You can easily convert between these two canonical forms by generating the truth table for one form and working backwards from the truth table to produce the other form. For example, consider the function of two variables, $F_7 = A + B$. The sum of minterms form is $F_7 = A'B + AB' + AB$. The truth table takes the form:

Table 16: F_7 (OR) Truth Table for Two Variables

F_7	A	B
0	0	0
0	1	0
1	0	1
1	1	1

Working backwards to get the product of maxterms, we locate all entries that have a zero result. This is the entry with A and B equal to zero. This gives us the first step of G=A'B'. However, we still need to invert all the variables to obtain G=AB. By the duality principle we need to swap the logical OR and logical AND operators obtaining G=A+B. This is the canonical *product of maxterms* form.

Since working with the product of maxterms is a little messier than working with sums of minterms, this text will generally use the sum of minterms form. Furthermore, the sum of minterms form is more common in boolean logic work. However, you will encounter both forms when studying logic design.

2.5 Simplification of Boolean Functions

Since there are an infinite variety of boolean functions of *n* variables, but only a finite number of unique boolean functions of those *n* variables, you might wonder if there is some method that will simplify a given boolean function to produce the optimal form. Of course, you can always use algebraic transformations to produce the optimal form, but using heuristics does not guarantee an optimal transformation. There are, however, two methods that *will* reduce a given boolean function to its optimal form: the map method and the prime implicants method. In this text we will only cover the mapping method, see any text on logic design for other methods.

Since for any logic function some optimal form must exist, you may wonder why we don't use the optimal form for the canonical form. There are two reasons. First, there may be several optimal forms. They are not guaranteed to be unique. Second, it is easy to convert between the canonical and truth table forms.

Using the map method to optimize boolean functions is practical only for functions of two, three, or four variables. With care, you can use it for functions of five or six variables, but the map method is cumbersome to use at that point. For more than six variables, attempting map simplifications by hand would not be wise[1].

The first step in using the map method is to build a two-dimensional truth table for the function (see Figure 2.1).

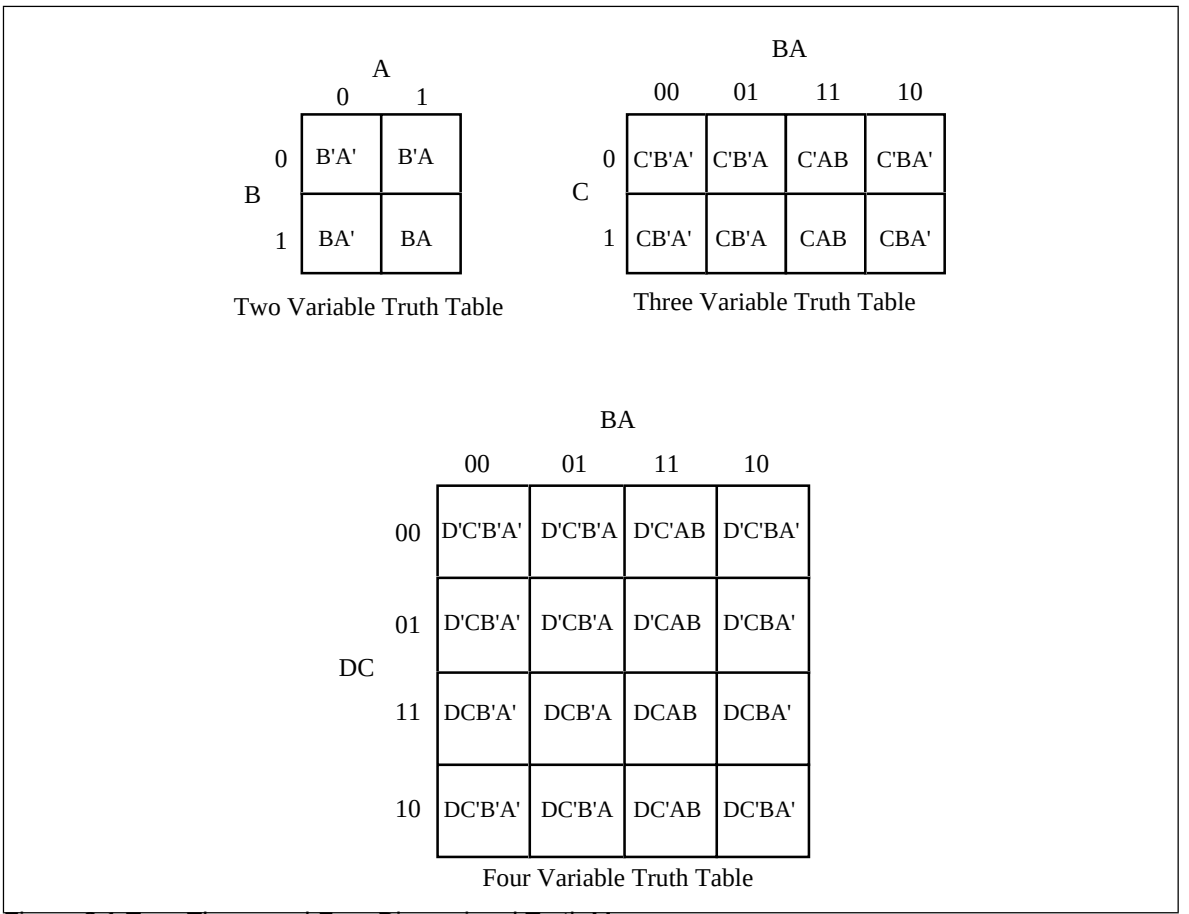


Figure 2.1 Two, Three, and Four Dimensional Truth Maps

Warning: Take a careful look at these truth tables. They do not use the same forms appearing earlier in this chapter. In particular, the progression of the values is 00, 01, 11, 10, not 00, 01, 10, 11. This is very important! If you organize the truth tables in a binary sequence, the mapping optimization method will not work properly. We will call this a *truth map* to distinguish it from the standard truth table.

Assuming your boolean function is in canonical form (sum of minterms), insert ones for each of the truth map entries corresponding to a minterm in the function. Place zeros everywhere else. For example, consider the function of three variables F=C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA. Figure 2.2 shows the truth map for this function.

1. However, it's probably quite reasonable to write a *program* that uses the map method for seven or more variables.

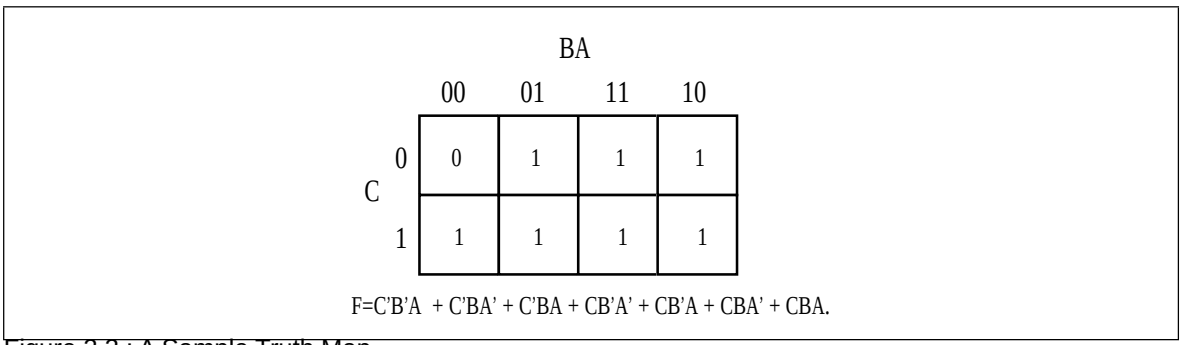


Figure 2.2 : A Sample Truth Map

The next step is to draw rectangles around rectangular groups of ones. The rectangles you enclose must have sides whose lengths are powers of two. For functions of three variables, the rectangles can have sides whose lengths are one, two, and four. The set of rectangles you draw must surround all cells containing ones in the truth map. The trick is to draw all possible rectangles unless a rectangle would be completely enclosed within another. Note that the rectangles may overlap if one does not enclose the other. In the truth map in Figure 2.2 there are three such rectangles (see Figure 2.3)

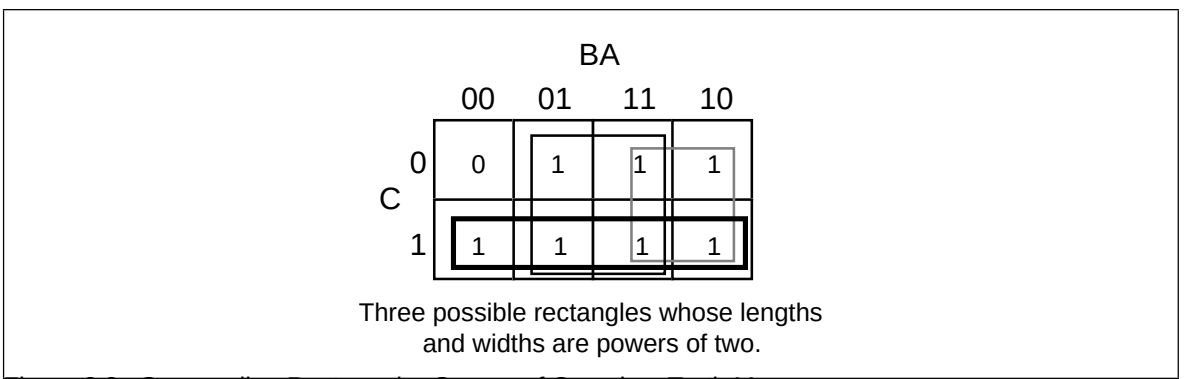


Figure 2.3 : Surrounding Rectangular Groups of Ones in a Truth Map

Each rectangle represents a term in the simplified boolean function. Therefore, the simplified boolean function will contain only three terms. You build each term using the process of elimination. You eliminate any variables whose primed and unprimed form both appear within the rectangle. Consider the long skinny rectangle above that is sitting in the row where C=1. This rectangle contains both A and B in primed and unprimed form. Therefore, we can eliminate A and B from the term. Since the rectangle sits in the C=1 region, this rectangle represents the single literal C.

Now consider the solid square above. This rectangle includes C, C', B, B' and A. Therefore, it represents the single term A. Likewise, the square with the dotted line above contains C, C', A, A' and B. Therefore, it represents the single term B.

The final, optimal, function is the sum (logical OR) of the terms represented by the three squares. Therefore, F= A + B + C. You do not have to consider squares containing zeros.

When enclosing groups of ones in the truth map, you must consider the fact that a truth map forms a *torus* (i.e., a doughnut shape). The right edge of the map *wraps around* to the left edge (and vice-versa). Likewise, the top edge *wraps around* to the bottom edge. This introduces additional possibilities when surrounding groups of ones in a map. Consider the boolean function F=C'B'A' + C'BA' + CB'A' + CBA'. Figure 2.4 shows the truth map for this function.

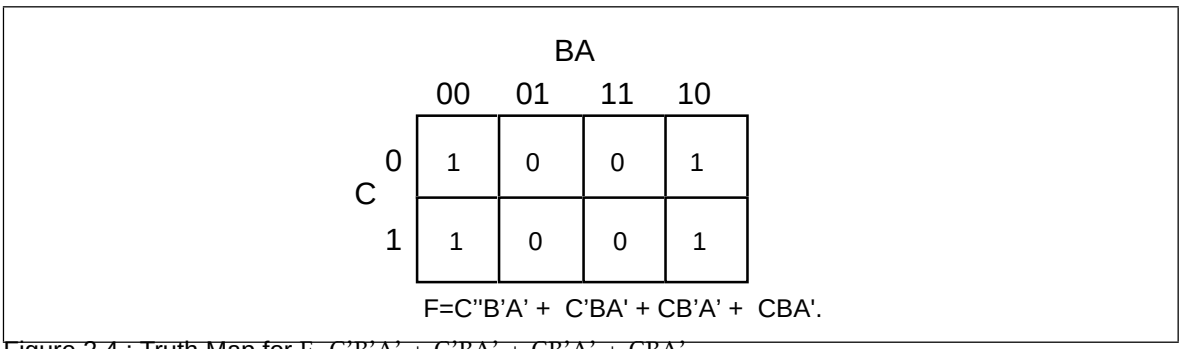


Figure 2.4 : Truth Map for F=C'B'A' + C'BA' + CB'A' + CBA'

At first glance, you would think that there are two possible rectangles here as Figure 2.5 shows. However, because the truth map is a continuous object with the right side and left sides connected, we can form a single, square rectangle, as Figure 2.6 shows.

So what? Why do we care if we have one rectangle or two in the truth map? The answer is because the larger the rectangles are, the more terms they will eliminate. The fewer rectangles that we have, the fewer terms will appear in the final boolean function. For example, the former example with two rectangles generates a function with two terms. The first rectangle (on the left) eliminates the C variable, leaving A'B' as its term. The second rectangle, on the right, also eliminates the C variable, leaving the term BA'. Therefore, this truth map would produce the equation F=A'B' + A'B. We know this is not optimal, see Th 13. Now consider the second truth map above. Here we have a single rectangle so our boolean function will only have a single term. Obviously this is more optimal than an equation with two terms. Since this rectangle includes both C and C' and also B and B', the only term left is A'. This boolean function, therefore, reduces to F=A'.

There are only two cases that the truth map method cannot handle properly: a truth map that contains all zeros or a truth map that contains all ones. These two cases correspond to the boolean functions F=0 and F=1, respectively. These functions are easy to generate by inspection of the truth map.

An important thing you must keep in mind when optimizing boolean functions using the mapping method is that you always want to pick the largest rectangles whose sides'

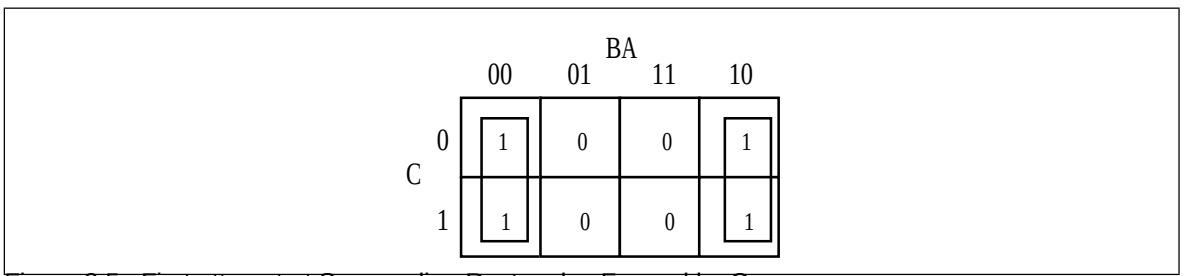


Figure 2.5 : First attempt at Surrounding Rectangles Formed by Ones

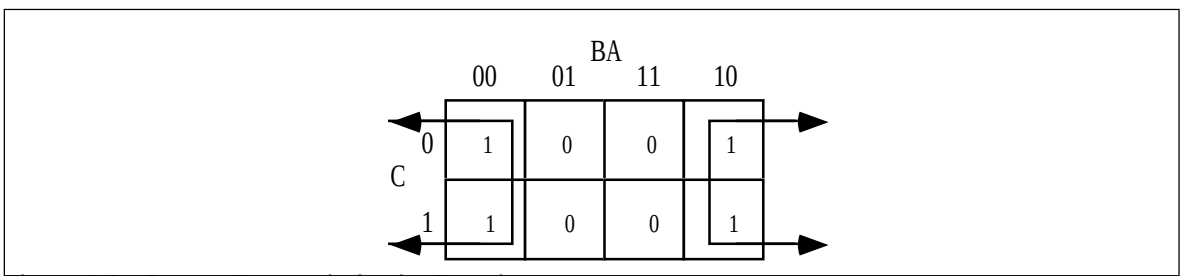


Figure 2.6 : Correct Rectangle for the Function

lengths are a power of two. You must do this even for overlapping rectangles (unless one rectangle encloses another). Consider the boolean function F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA. This produces the truth map appearing in Figure 2.7.

The initial temptation is to create one of the sets of rectangles found in Figure 2.8. However, the correct mapping appears in Figure 2.9.

All three mappings will produce a boolean function with two terms. However, the first two will produce the expressions F= B + A'B' and F = AB + A'. The third form produces F = B + A'. Obviously, this last form is more optimal than the other two forms (see theorems 11 and 12).

For functions of three variables, the size of the rectangle determines the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have three literals.
- A rectangle surrounding two squares containing ones represents a term containing two literals.
- A rectangle surrounding four squares containing ones represents a term containing a single literal.
- A rectangle surrounding eight squares represents the function F = 1.

Truth maps you create for functions of four variables are even trickier. This is because there are lots of places rectangles can hide from you along the edges. Figure 2.10 shows some possible places rectangles can hide.

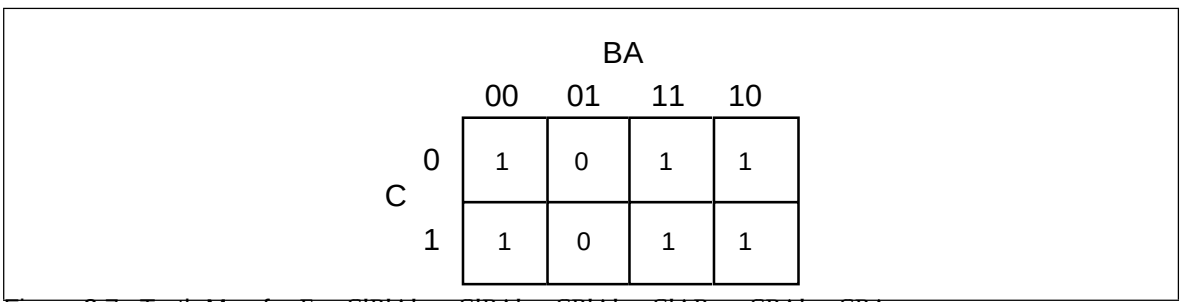


Figure 2.7 : Truth Map for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

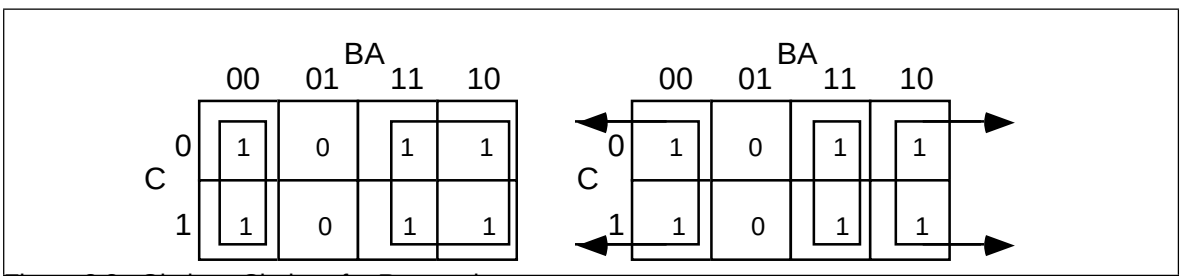


Figure 2.8 : Obvious Choices for Rectangles

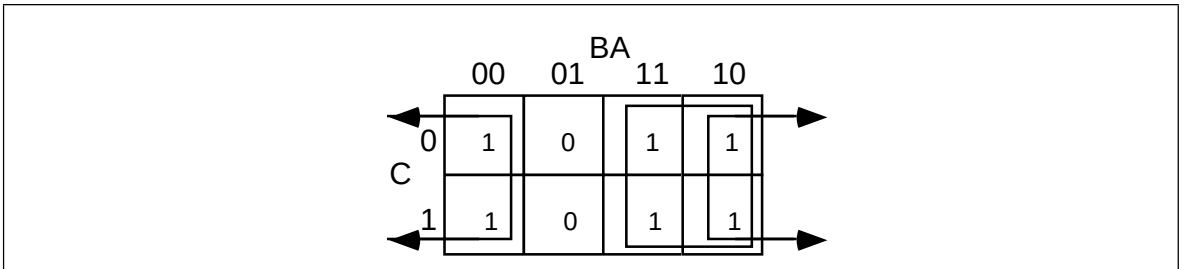


Figure 2.9 Correct Set of Rectangles for F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA

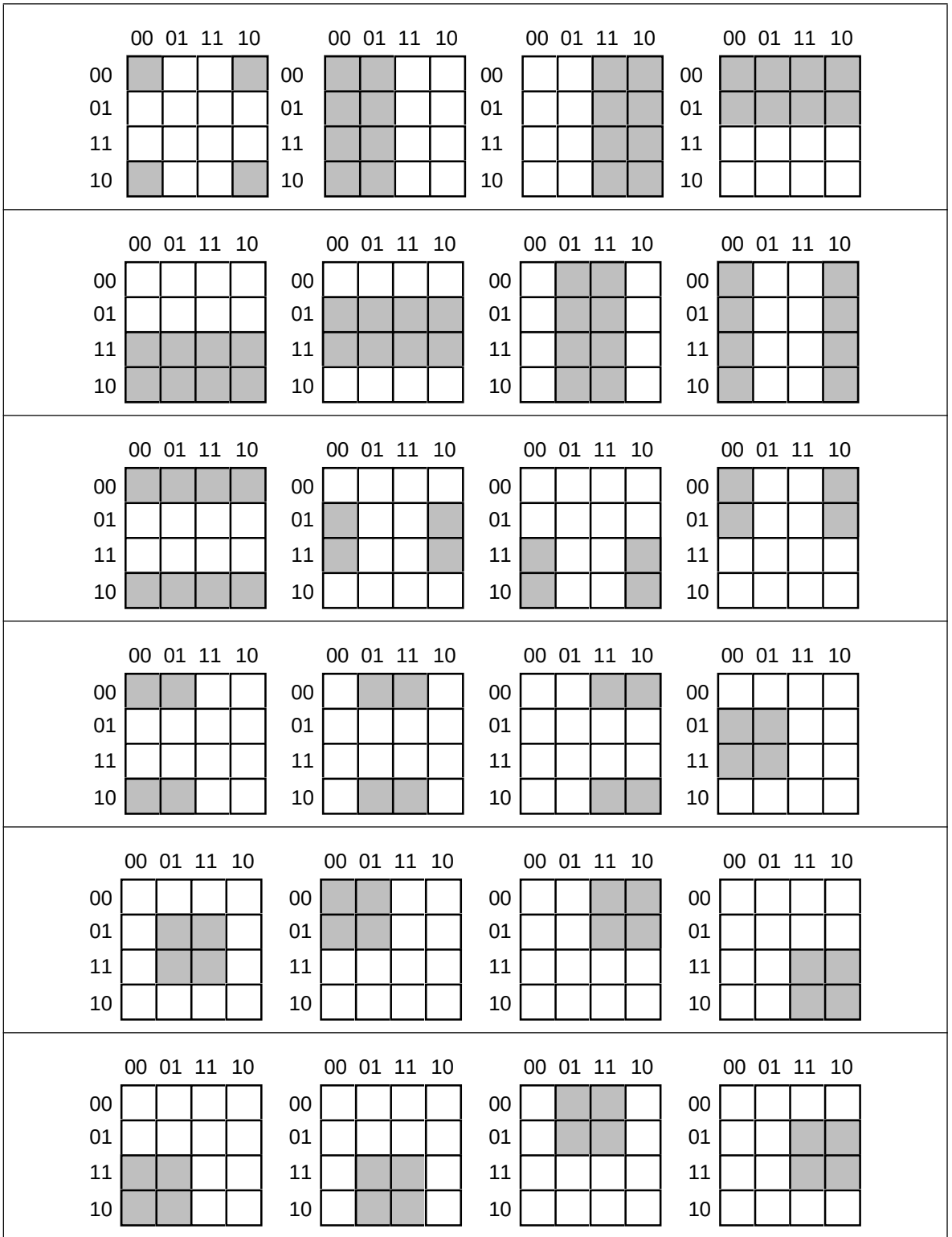


Figure 2.10 : Partial Pattern List for 4x4 Truth Map

This list of patterns doesn't even begin to cover all of them! For example, these diagrams show none of the 1x2 rectangles. You must exercise care when working with four variable maps to ensure you select the largest possible rectangles, especially when overlap occurs. This is particularly important with you have a rectangle next to an edge of the truth map.

As with functions of three variables, the size of the rectangle in a four variable truth map controls the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have four literals.
- A rectangle surrounding two squares containing ones represents a term containing three literals.
- A rectangle surrounding four squares containing ones represents a term containing two literals.
- A rectangle surrounding eight squares containing ones represents a term containing a single literal.
- A rectangle surrounding sixteen squares represents the function F=1.

This last example demonstrates an optimization of a function containing four variables. The function is F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA', the truth map appears in Figure 2.11.

Here are two possible sets of maximal rectangles for this function, each producing three terms (see Figure 2.12). Both functions are equivalent; both are as optimal as you can get[2]. Either will suffice for our purposes.

First, let's consider the term represented by the rectangle formed by the four corners. This rectangle contains B, B', D, and D'; so we can eliminate those terms. The remaining terms contained within these rectangles are C' and A', so this rectangle represents the term C'A'.

The second rectangle, common to both maps in Figure 2.12, is the rectangle formed by the middle four squares. This rectangle includes the terms A, B, B', C, D, and D'. Eliminating B, B', D, and D' (since both primed and unprimed terms exist), we obtain CA as the term for this rectangle.

The map on the left in Figure 2.12 has a third term represented by the top row. This term includes the variables A, A', B, B', C' and D'. Since it contains A, A', B, and B', we can

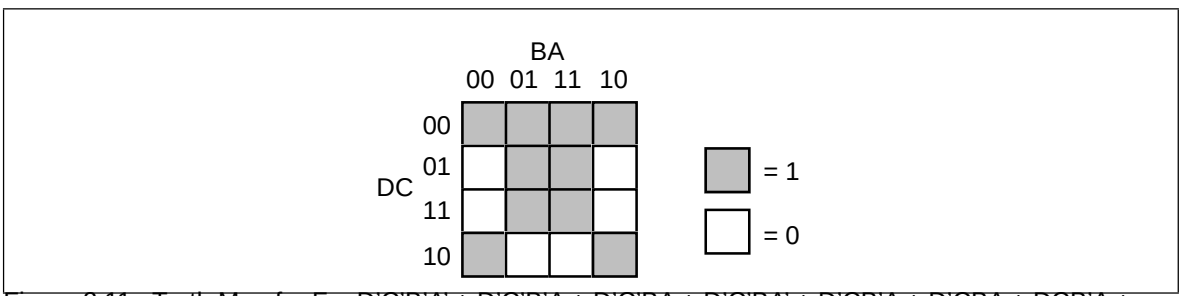


Figure 2.11 : Truth Map for F = D'C'B'A' + D'C'B'A + D'C'BA + D'C'BA' + D'CB'A + D'CBA + DCB'A + DCBA + DC'B'A' + DC'BA'

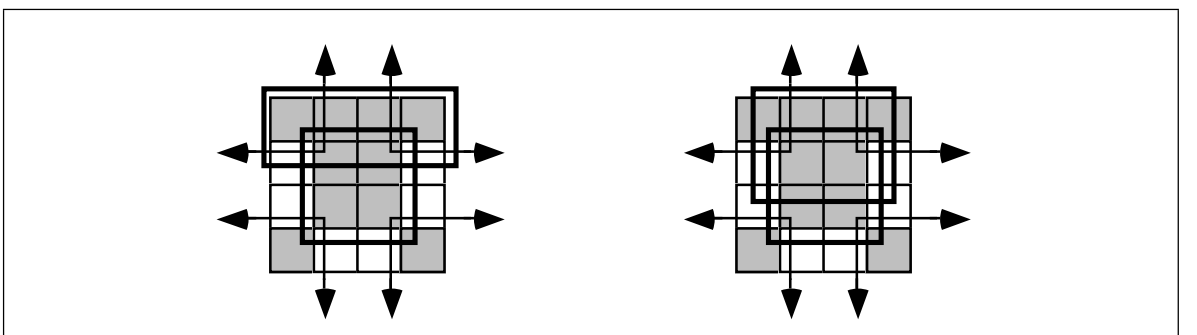


Figure 2.12 : Two Combinations of Surrounded Values Yielding Three Terms

2. Remember, there is no guarantee that there is a unique optimal solution.

eliminate these terms. This leaves the term C'D'. Therefore, the function represented by the map on the left is F=C'A' + CA + C'D'.

The map on the right in Figure 2.12 has a third term represented by the top/middle four squares. This rectangle subsumes the variables A, B, B', C, C', and D'. We can eliminate B, B', C, and C' since both primed and unprimed versions appear, this leaves the term AD. Therefore, the function represented by the function on the right is F=C'A' + CA + AD'.

Since both expressions are equivalent, contain the same number of terms, and the same number of operators, either form is equivalent. Unless there is another reason for choosing one over the other, you can use either form.

2.6 What Does This Have To Do With Computers, Anyway?

Although there is a tenuous relationship between boolean functions and boolean expressions in programming languages like C or Pascal, it is fair to wonder why we're spending so much time on this material. However, the relationship between boolean logic and computer systems is much stronger. There is a one-to-one relationship between boolean functions and electronic circuits. Electrical engineers who design CPUs and other computer related circuits need to be intimately familiar with this stuff. Even if you never intend to design your own electronic circuits, understanding this relationship is important if you want to make the most of any computer system.

2.6.1 Correspondence Between Electronic Circuits and Boolean Functions

There is a one-to-one correspondence between an electrical circuits and boolean functions. For any boolean function you can design an electronic circuit and vice versa. Since boolean functions only require the AND, OR, and NOT boolean operators, we can construct any electronic circuit using these operations exclusively. The boolean AND, OR, and NOT functions correspond to the following electronic circuits, the AND, OR, and inverter (NOT) gates (see Figure 2.13).

One interesting fact is that you only need a single gate type to implement *any* electronic circuit. This gate is the *NAND* gate, shown in Figure 2.14.

To prove that we can construct any boolean function using only NAND gates, we need only show how to build an inverter (NOT), AND gate, and OR gate from a NAND (since we can create any boolean function using only AND, NOT, and OR). Building an inverter is easy, just connect the two inputs together (see Figure 2.15).

Once we can build an inverter, building an AND gate is easy – just invert the output of a NAND gate. After all, NOT (NOT (A AND B)) is equivalent to A AND B (see). Of course, this takes *two* NAND gates to construct a single AND gate, but no one said that

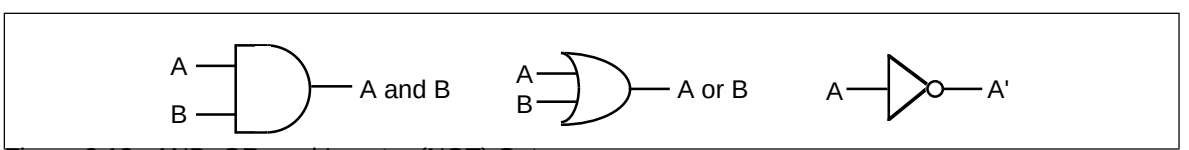


Figure 2.13 : AND, OR, and Inverter (NOT) Gates

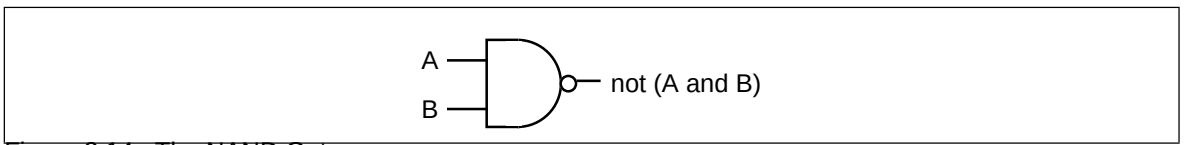


Figure 2.14 : The NAND Gate

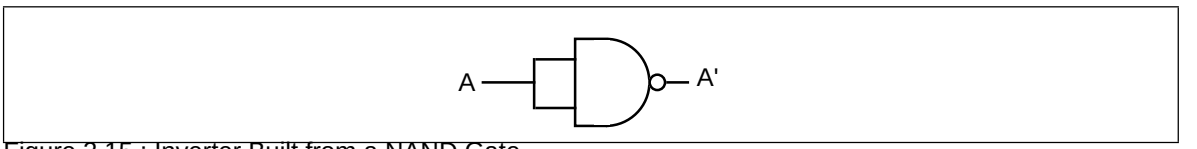


Figure 2.15 : Inverter Built from a NAND Gate

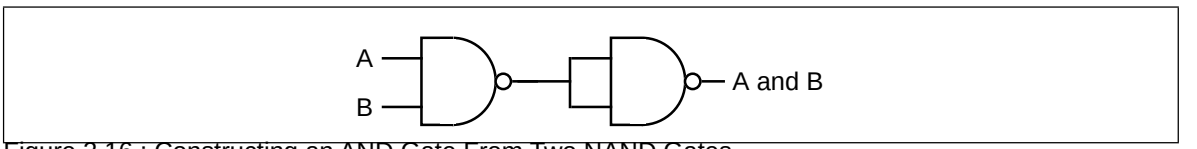


Figure 2.16 : Constructing an AND Gate From Two NAND Gates

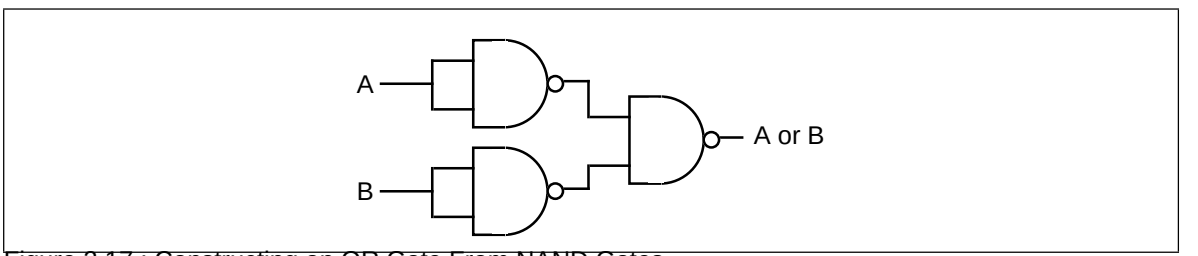


Figure 2.17 : Constructing an OR Gate From NAND Gates

circuits constructed only with NAND gates would be optimal, only that it is possible to do.

The remaining gate we need to synthesize is the logical-OR gate. We can easily construct an OR gate from NAND gates by applying DeMorgan's theorems.

```
(A or B)'     =         A' and B'         DeMorgan's Theorem.
A or B        =         (A' and B')'      Invert both sides of the equation.
A or B        =         A' nand B'        Definition of NAND operation.
```

By applying these transformations, you get the circuit in Figure 2.17.

Now you might be wondering why we would even bother with this. After all, why not just use logical AND, OR, and inverter gates directly? There are two reasons for this. First, NAND gates are generally less expensive to build than other gates. Second, it is also much easier to build up complex integrated circuits from the same basic building blocks than it is to construct an integrated circuit using different basic gates.

Note, by the way, that it is possible to construct any logic circuit using only NOR gates[3]. The correspondence between NAND and NOR logic is orthogonal to the correspondence between the two canonical forms appearing in this chapter (sum of minterms vs. product of maxterms). While NOR logic is useful for many circuits, most electronic designs use NAND logic. See the exercises for more examples.

2.6.2 Combinatorial Circuits

A combinatorial circuit is a system containing basic boolean operations (AND, OR, NOT), some inputs, and a set of outputs. Since each output corresponds to an individual logic function, a combinatorial circuit often implements several different boolean functions. It is very important that you remember this fact – each output represents a different boolean function.

A computer's CPU is built up from various combinatorial circuits. For example, you can implement an addition circuit using boolean functions. Suppose you have two one-bit

3. NOR is NOT (A OR B).

numbers, A and B. You can produce the one-bit sum and the one-bit carry of this addition using the two boolean functions:

```
S =   AB' + A'B                 Sum of A and B.
C =   AB                        Carry from addition of A and B.
```

These two boolean functions implement a *half-adder*. Electrical engineers call it a half adder because it adds two bits together but cannot add in a carry from a previous operation. A *full adder* adds three one-bit inputs (two bits plus a carry from a previous addition) and produces two outputs: the sum and the carry. The two logic equations for a full adder are

$$S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Although these logic equations only produce a single bit result (ignoring the carry), it is easy to construct an n-bit sum by combining adder circuits (see Figure 2.18). So, as this example clearly illustrates, we can use logic functions to implement arithmetic and boolean operations.

Another common combinatorial circuit is the *seven-segment decoder*. This is a combinatorial circuit that accepts four inputs and determines which of the seven segments on a seven-segment LED display should be on (logic one) or off (logic zero). Since a seven segment display contains seven output values (one for each segment), there will be seven logic functions associated with the display (segment zero through segment six). See Figure 2.19 for the segment assignments. Figure 2.20 shows the segment assignments for each of the ten decimal values.

The four inputs to each of these seven boolean functions are the four bits from a binary number in the range 0..9. Let D be the H.O. bit of this number and A be the L.O. bit of this number. Each logic function should produce a one (segment on) for a given input if that particular segment should be illuminated. For example S_4 (segment four) should be

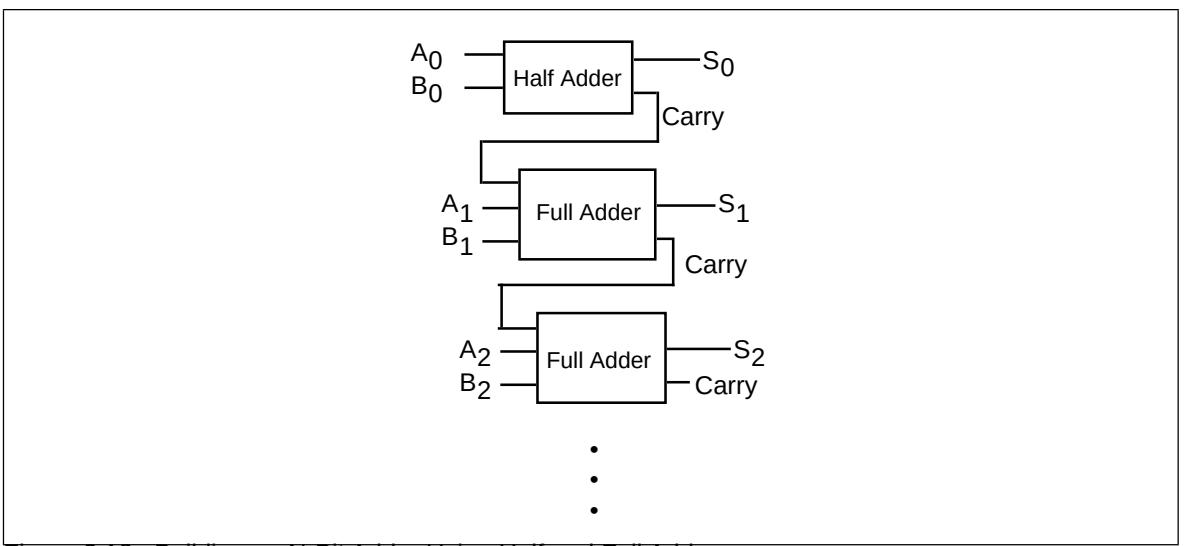


Figure 2.18 : Building an N-Bit Adder Using Half and Full Adders

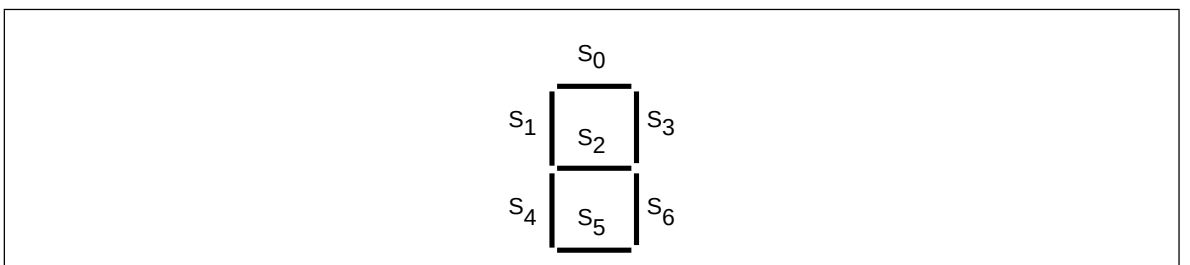


Figure 2.19 : Seven Segment Display

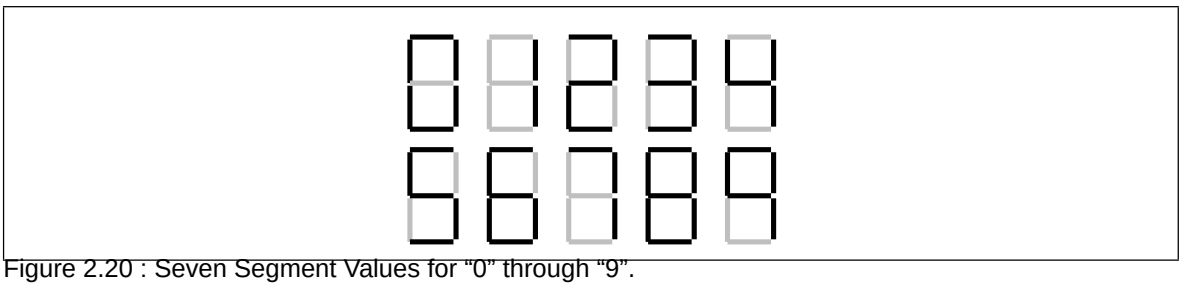


Figure 2.20 : Seven Segment Values for “0” through “9”.

on for binary values 0000, 0010, 0110, and 1000. For each value that illuminates a segment, you will have one minterm in the logic equation:

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'.$$

S_0, as a second example, is on for values zero, two, three, five, six, seven, eight, and nine. Therefore, the logic function for S_0 is

$$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A$$

You can generate the other five logic functions in a similar fashion (see the exercises).

Combinatorial circuits are the basis for many components of a basic computer system. You can construct circuits for addition, subtraction, comparison, multiplication, division, and many other operations using combinatorial logic.

2.6.3 Sequential and Clocked Logic

One major problem with combinatorial logic is that it is *memoryless*. In theory, all logic function outputs depend only on the current inputs. Any change in the input values is immediately reflected in the outputs[4]. Unfortunately, computers need the ability to *remember* the results of past computations. This is the domain of sequential or clocked logic.

A *memory cell* is an electronic circuit that remembers an input value after the removal of that input value. The most basic memory unit is the *set/reset flip-flop*. You can construct an *SR flip-flop* using two NAND gates, as shown in Figure 2.21.

The S and R inputs are normally high. If you *temporarily* set the S input to zero and then bring it back to one (*toggle* the S input), this forces the Q output to one. Likewise, if you toggle the R input from one to zero back to one, this sets the Q output to zero. The Q’ input is generally the inverse of the Q output.

Note that if both S and R are one, then the Q output depends upon Q. That is, whatever Q happens to be, the top NAND gate continues to output that value. If Q was originally one, then there are two ones as inputs to the bottom flip-flop (Q nand R). This

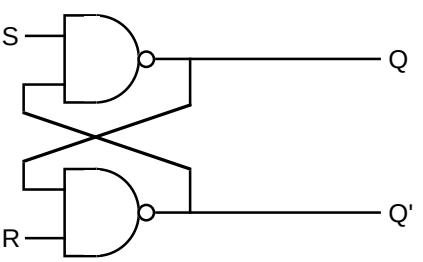


Figure 2.21 : Set/Reset Flip Flop Constructed From NAND Gates

4. In practice, there is a short *propagation delay* between a change in the inputs and the corresponding outputs in any electronic implementation of a boolean function.

produces an output of zero (Q'). Therefore, the two inputs to the top NAND gate are zero and one. This produces the value one as an output (matching the original value for Q).

If the original value for Q was zero, then the inputs to the bottom NAND gate are Q=0 and R=1. Therefore, the output of this NAND gate is one. The inputs to the top NAND gate, therefore, are S=1 and Q'=1. This produces a zero output, the original value of Q.

Suppose Q is zero, S is zero and R is one. This sets the two inputs to the top flip-flop to one and zero, forcing the output (Q) to one. Returning S to the high state does not change the output at all. You can obtain this same result if Q is one, S is zero, and R is one. Again, this produces an output value of one. This value remains one even when S switches from zero to one. Therefore, toggling the S input from one to zero and then back to one produces a one on the output (i.e., *sets* the flip-flop). The same idea applies to the R input, except it forces the Q output to zero rather than to one.

There is one catch to this circuit. It does not operate properly if you set both the S and R inputs to zero simultaneously. This forces both the Q and Q' outputs to one (which is logically inconsistent). Whichever input remains zero the longest determines the final state of the flip-flop. A flip-flop operating in this mode is said to be *unstable.*

The only problem with the S/R flip-flop is that you must use separate inputs to remember a zero or a one value. A memory cell would be more valuable to us if we could specify the data value to remember on one input and provide a *clock input* to *latch* the input value. This type of flip-flop, the D flip-flop (for *data*) uses the circuit in Figure 2.22.

Assuming you fix the Q and Q' outputs to either 0/1 or 1/0, sending a *clock pulse* that goes from zero to one back to zero will copy the D input to the Q output. It will also copy D' to Q'. The exercises at the end of this chapter will expect you to describe this operation in detail, so study this diagram carefully.

Although remembering a single bit is often important, in most computer systems you will want to remember a group of bits. You can remember a sequence of bits by combining several D flip-flops in parallel. Concatenating flip-flops to store an n-bit value forms a *register.* The electronic schematic in Figure 2.23 shows how to build an eight-bit register from a set of D flip-flops.

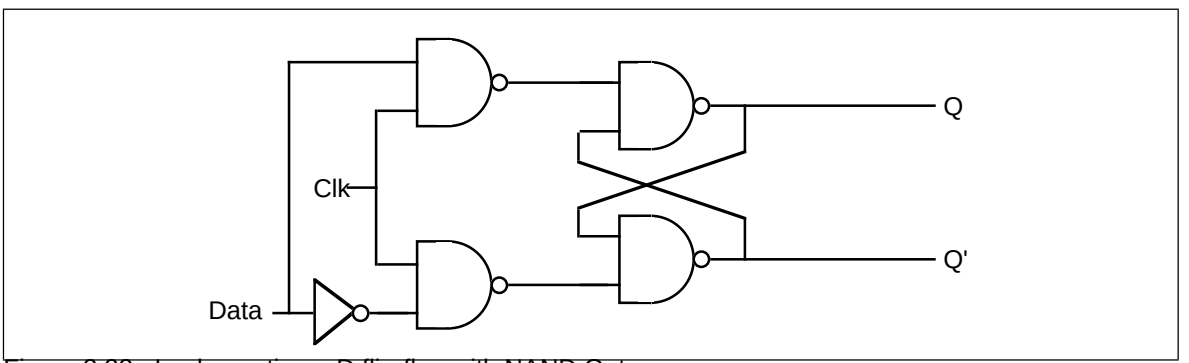


Figure 2.22 : Implementing a D flip-flop with NAND Gates

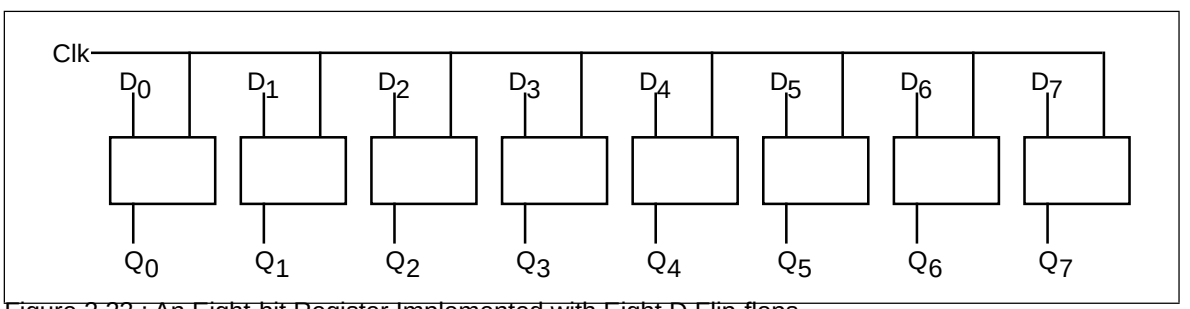


Figure 2.23 : An Eight-bit Register Implemented with Eight D Flip-flops

Note that the eight D flip-flops use a common clock line. This diagram does not show the Q' outputs on the flip-flops since they are rarely required in a register.

D flip-flops are useful for building many sequential circuits above and beyond simple registers. For example, you can build a *shift register* that shifts the bits one position to the left on each clock pulse. A four-bit shift register appears in Figure 2.24.

You can even build a *counter,* that counts the number of times the clock toggles from one to zero and back to one using flip-flops. The circuit in Figure 2.25 implements a four bit counter using D flip-flops.

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits beyond these.

2.7 Okay, What Does It Have To Do With Programming, Then?

Once you have registers, counters, and shift registers, you can build *state machines.* The implementation of an algorithm in hardware using state machines is well beyond the scope of this text. However, one important point must be made with respect to such circuitry – *any algorithm you can implement in software you can also implement directly in hardware.* This suggests that boolean logic is the basis for computation on all modern computer systems. Any program you can write, you can specify as a sequence of boolean equations.

Of course, it is much easier to specify a solution to a programming problem using languages like Pascal, C, or even assembly language than it is to specify the solution using boolean equations. Therefore, it is unlikely that you would ever implement an entire program using a set of state machines and other logic circuitry. Nevertheless, there are times when a hardware implementation is better. A hardware solution can be one, two, three, or more *orders of magnitude* faster than an equivalent software solution. Therefore, some time critical operations may require a hardware solution.

A more interesting fact is that the converse of the above statement is also true. Not only can you implement all software functions in hardware, but it is also possible to *implement all hardware functions in software.* This is an important revelation because many operations you would normally implement in hardware are *much cheaper* to implement using software on a microprocessor. Indeed, this is a primary use of *assembly language* in modern

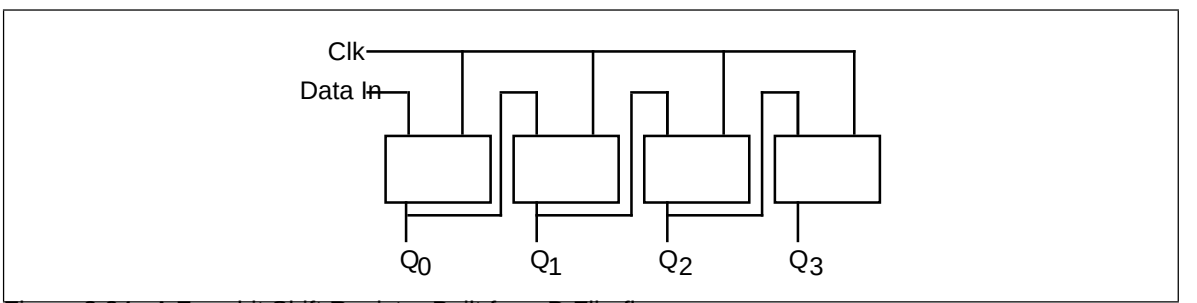


Figure 2.24 : A Four-bit Shift Register Built from D Flip-flops

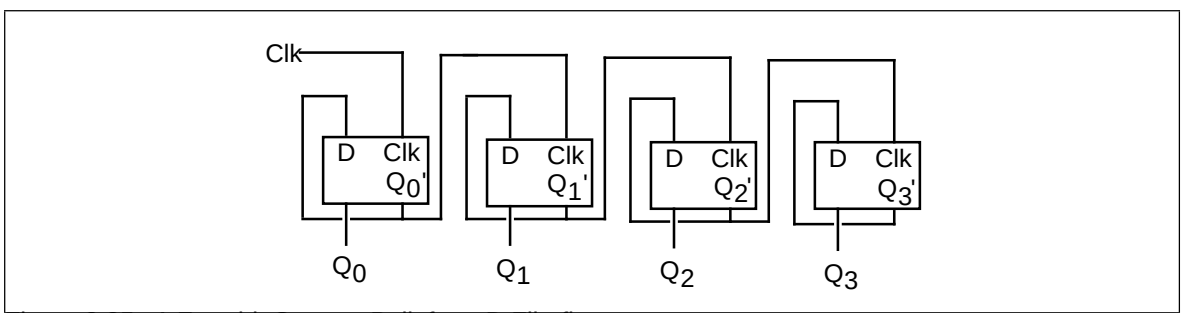


Figure 2.25 : A Four-bit Counter Built from D Flip-flops

systems – to inexpensively replace a complex electronic circuit. It is often possible to replace many tens or hundreds of dollars of electronic components with a single $25 microcomputer chip. The whole field of *embedded systems* deals with this very problem. Embedded systems are computer systems embedded in other products. For example, most microwave ovens, TV sets, video games, CD players, and other consumer devices contain one or more complete computer systems whose sole purpose is to replace a complex hardware design. Engineers use computers for this purpose because they are *less expensive* and *easier to design with* than traditional electronic circuitry.

You can easily design software that reads switches (input variables) and turns on motors, LEDs or lights, locks or unlocks a door, etc. (output functions). To write such software, you will need an understanding of boolean functions and how to implement such functions in software.

Of course, there is one other reason for studying boolean functions, even if you never intend to write software intended for an embedded system or write software that manipulates real-world devices. Many high level languages process boolean expressions (e.g., those expressions that control an if statement or while loop). By applying transformations like DeMorgan's theorems or a mapping optimization it is often possible to improve the performance of high level language code. Therefore, studying boolean functions *is* important even if you never intend to design an electronic circuit. It can help you write better code in a traditional programming language.

For example, suppose you have the following statement in Pascal:

```
if ((x=y) and (a <> b)) or ((x=y) and (c <= d)) then SomeStmt;
```

You can use the distributive law to simplify this to:

```
if ((x=y) and ((a <> b) or (c <= d)) then SomeStmt;
```

Likewise, we can use DeMorgan's theorem to reduce

```
while (not((a=b) and (c=d)) do Something;
```

to

```
while (a <> b) or (c <> d) do Something;
```

2.8 Generic Boolean Functions

For a specific application, you can create a logic function that achieves some specific result. Suppose, however, that you wanted to write a program to *simulate* any possible boolean function? For example, on the companion diskette, there is a program that lets you enter an arbitrary boolean function with one to four different variables. This program will read the inputs and produce and necessary function results. Since the number of unique four variable functions is large (65,536, to be exact), it is not practical to include a specific solution for each one in a program. What is necessary is a *generic logic function*, one that will compute the results for any arbitrary function. This section describes how to write such a function.

A generic boolean function of four variables requires five parameters – the four input parameters and a fifth parameter that specifies the function to compute. While there are lots of ways to specify the function to compute, we'll pass the boolean function's number as this fifth parameter.

At first glance you might wonder how we can compute a function using the function's number. However, keep in mind that the bits that make up the function's number come directly from the truth table for that function. Therefore, if we extract the bits from the function's number, we can construct the truth table for that function. Indeed, if we just select the ith bit of the function number, where i = D*8 + C*4 + B*2 +A you will get the

function result for that particular value of A, B, C, and D[5]. The following examples, in C and Pascal, show how to write such functions:

```
/************************************************************************/
/*                                                                      */
/* This C program demonstrates how to write a generic logic function    */
/* that can compute any logic function of four variables.  Given C's    */
/* bit manipulation operators, along with hexadecimal I/O, this is an   */
/* easy task to accomplish in the C programming language.               */
/*                                                                      */
/************************************************************************/

#include <stdlib.h>
#include <stdio.h>


/* Generic logic function.  The "Func" parameter contains the 16-bit    */
/* logical function number.  This is actually an encoded truth table    */
/* for the function.  The a, b, c, and d parameters are the inputs to   */
/* the logic function.  If we treat "func" as a 2x2x2x2 array of bits,  */
/* this particular function selects bit "func[d,c,b,a]" from func.      */

int
generic(int func, int a, int b, int c, int d)
{
        /* Return the bit specified by a, b, c, and d */

        return (func >> (a + b*2 + c*4 + d*8)) & 1;
}




/* Main program to drive the generic logic function written in C.       */

main()
{
        int func, a, b, c, d;

        /* Repeat the following until the user enters zero.             */

        do
        {
                /* Get the function's number (truth table)              */

                printf("Enter function value (hex): ");
                scanf("%x", &func);

                /* If the user specified zero as the function #, stop   */
                /* the program.                                         */

                if (func != 0)
                {
                        printf("Enter values for d, c, b, & a: ");
                        scanf("%d%d%d%d",
                                &d, &c, &b, &a);

                        printf("The result is %d\n", generic(func,a,b,c,d));
                        printf("Func = %x, A=%d, B=%d, C=%d, D=%d\n",
                                func, a, b, c, d);
                }


        } while (func !=0);

}
```

The following Pascal program is written for *Standard Pascal*. Standard Pascal does not provide any bit manipulation operations, so this program is lengthy since it has to simulate bits using an array of integers. Most modern Pascals (especially Turbo Pascal) provide built-in bit operations or library routines that operate on bits. This program would be much easier to write using such non-standard features.

5. Chapter Five explains why this multiplication works.

```
program GenericFunc(input,output);

(* Since standard Pascal does not provide an easy way to directly man-  *)
(* ipulate bits in an integer, we will simulate the function number     *)
(* using an array of 16 integers. “GFTYPE” is the type of that array.   *)

type
    gftype = array [0..15] of integer;

var
   a, b, c, d:integer;
   fresult:integer;
   func: gftype;


(* Standard Pascal does not provide the ability to shift integer data   *)
(* to the left or right.  Therefore, we will simulate a 16-bit value    *)
(* using an array of 16 integers.  We can simulate shifts by moving     *)
(* data around in the array.                                            *)
(*                                                                      *)
(* Note that Turbo Pascal *does* provide shl and shr operators.  How-   *)
(* ever, this code is written to work with standard Pascal, not just    *)
(* Turbo Pascal.                                                        *)
(*                                                                      *)
(* ShiftLeft shifts the values in func on position to the left and in-  *)
(* serts the shiftin value into “bit position” zero.                    *)

procedure ShiftLeft(shiftin:integer);
var i:integer;
begin

     for i := 15 downto 1 do func[i] := func[i-1];
     func[0] := shiftin;

end;


(* ShiftNibble shifts the data in func to the left four positions and   *)
(* inserts the four bits a (L.O.), b, c, and d (H.O.) into the vacated  *)
(* positions.                                                           *)

procedure ShiftNibble(d,c,b,a:integer);
begin

     ShiftLeft(d);
     ShiftLeft(c);
     ShiftLeft(b);
     ShiftLeft(a);
end;


(* ShiftRight shifts the data in func one position to the right.  It    *)
(* shifts a zero into the H.O. bit of the array.                        *)

procedure ShiftRight;
var i:integer;
begin

     for i := 0 to 14 do func[i] := func[i+1];
     func[15] := 0;

end;


(* ToUpper converts a lower case character to upper case.               *)

procedure toupper(var ch:char);
begin

     if (ch in [‘a’..’z’]) then ch := chr(ord(ch) - 32);

end;


(* ReadFunc reads a hexadecimal function number from the user and puts  *)
(* this value into the func array (bit by bit).                         *)

function ReadFunc:integer;
```

```
var ch:char;
    i, val:integer;
begin

     write('Enter function number (hexadecimal): ');
     for i := 0 to 15 do func[i] := 0;
     repeat

           read(ch);
           if not eoln then begin

                     toupper(ch);
                     case ch of
                          '0': ShiftNibble(0,0,0,0);
                          '1': ShiftNibble(0,0,0,1);
                          '2': ShiftNibble(0,0,1,0);
                          '3': ShiftNibble(0,0,1,1);
                          '4': ShiftNibble(0,1,0,0);
                          '5': ShiftNibble(0,1,0,1);
                          '6': ShiftNibble(0,1,1,0);
                          '7': ShiftNibble(0,1,1,1);
                          '8': ShiftNibble(1,0,0,0);
                          '9': ShiftNibble(1,0,0,1);
                          'A': ShiftNibble(1,0,1,0);
                          'B': ShiftNibble(1,0,1,1);
                          'C': ShiftNibble(1,1,0,0);
                          'D': ShiftNibble(1,1,0,1);
                          'E': ShiftNibble(1,1,1,0);
                          'F': ShiftNibble(1,1,1,1);
                          else write(chr(7),chr(8));
                     end;
           end;
     until eoln;
     val := 0;
     for i := 0 to 15 do val := val + func[i];
     ReadFunc := val;
end;


(* Generic - Computes the generic logical function specified by *)
(*           the function number "func" on the four input vars  *)
(*           a, b, c, and d.  It does this by returning bit     *)
(*           d*8 + c*4 + b*2 + a from func.                     *)

function Generic(var func:gftype; a,b,c,d:integer):integer;
begin
          Generic := func[a + b*2 + c*4 + d*8];
end;


begin (* main *)

      repeat

            fresult := ReadFunc;
            if (fresult <> 0) then begin

               write('Enter values for D, C, B, & A (0/1):');
               readln(d, c, b, a);
               writeln('The result is ',Generic(func,a,b,c,d));

            end;
      until fresult = 0;

end.
```

The following code demonstrates the power of bit manipulation operations. This version of the code above uses special features present in the Turbo Pascal programming language that allows programmers to shift left or right and do a bitwise logical AND on integer variables:

```
program GenericFunc(input,output);
const
     hex = ['a'..'f', 'A'..'F'];
     decimal = ['0'..'9'];

var
```

```
   a, b, c, d:integer;
   fresult:integer;
   func: integer;


(* Here is a second version of the Pascal generic function that uses *)
(* the features of Turbo Pascal to simplify the program.             *)


function ReadFunc:integer;
var ch:char;
    i, val:integer;
begin

     write('Enter function number (hexadecimal): ');
     repeat

           read(ch);
           func := 0;
           if not eoln then begin

              if (ch in Hex) then
                func := (func shl 4) + (ord(ch) and 15) + 9
              else if (ch in Decimal) then
                func := (func shl 4) + (ord(ch) and 15)
              else write(chr(7));

           end;
     until eoln;
     ReadFunc := func;
end;


(* Generic - Computes the generic logical function specified by *)
(*           the function number "func" on the four input vars  *)
(*           a, b, c, and d.  It does this by returning bit     *)
(*           d*8 + c*4 + b*2 + a from func.  This version re-   *)
(*           lies on Turbo Pascal's shift right operator and    *)
(*           its ability to do bitwise operations on integers.  *)

function Generic(func,a,b,c,d:integer):integer;
begin
        Generic := (func shr (a + b*2 + c*4 + d*8)) and 1;
end;


begin (* main *)

      repeat

            fresult := ReadFunc;
            if (fresult <> 0) then begin

               write('Enter values for D, C, B, & A (0/1):');
               readln(d, c, b, a);
               writeln('The result is ',Generic(func,a,b,c,d));

            end;
      until fresult = 0;

end.
```

2.9 Laboratory Exercises

This laboratory uses several Windows programs to manipulate truth tables and logic expressions, optimize logic equations, and simulate logic equations. These programs will help you understand the relationship between logic equations and truth tables as well as gain a fuller understanding of logic systems.

The *WLOGIC.EXE* program simulates logic circuitry. WLOGIC stores several logic equations that describe an electronic circuit and then it simulates that circuit using "switches" as inputs and "LEDs" as outputs. For those who would like a more "real-world" laboratory, there is an optional program you can run from DOS,

LOGICEV.EXE, that controls a real set of LEDs and switches that you construct and attach to the PC's parallel port. The directions for constructing this hardware appear in the appendices. The use of *either* program will let you easily observe the behavior of a set of logic functions.

If you haven't done so already, please install the software for this text on your machine. See the laboratory exercises in Chapter One for more details.

2.9.1 Truth Tables and Logic Equations Exercises

In this laboratory exercise you will create several different truth tables of two, three, and four variables. The TRUTHTBL.EXE program (found in the CH2 subdirectory) will automatically convert the truth tables you input into logic equations in the sum of minterms canonical form.

The TRUTHTBL.EXE file is a Windows program; it requires some version of Windows for proper operation. In particular, it will not run properly from DOS. It should, however, work just fine with any version of Windows from Windows 3.1 on up.

The TRUTHTBL.EXE program provides three buttons that let you choose a two variable, three variable, or four variable truth table. Pressing one of these buttons rearranges the truth table in an appropriate fashion. By default, the TRUTHTBL program assumes you want to work with a four variable truth table. Try pressing the *Two Variables*, *Three Variables*, and *Four Variables* buttons and observe the results. Describe what happens in your lab report.

To change the truth table entries, all you need do is click on the square associated with the truth table value you want to change. Clicking on one of these boxes toggles (inverts) that value in that square. For example, try clicking on the DCBA square several times and observe the results.

Note that as you click on different truth table entries, the TRUTHTBL program automatically recomputes the sum of minterms canonical logic equation and displays it at the bottom of the window. What equation does the program display if you set all squares in the truth table to zero?[6]

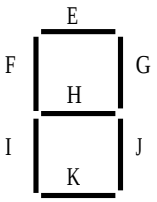
Set up the TRUTHTBL program to work with four variables. Set the DCBA square to one. Now press the *Two Variables* button. Press the *Four Variables* button and set *all* the squares to one. Now press the *Two Variables* button again. Finally, press the *Four Variables* button and examine the results. What does the TRUTHTBL program do when you switch between different sized truth tables? Feel free to try additional experiments to verify your hypothesis. Describe your results in your lab report.

Switch to two variable mode. Input the truth tables for the logical AND, OR, XOR, and NAND truth tables. Verify the correctness of the resulting logic equations. Write up the results in your lab report. Note: if there is a Windows-compatible printer attached to your computer, you can print each truth table you create by pressing the Print button in the window. This makes it very easy to include the truth table and corresponding logic equation in your lab report. **For additional credit:** input the truth tables for all 16 functions of two variables. In your lab report, present the results for these 16 functions.

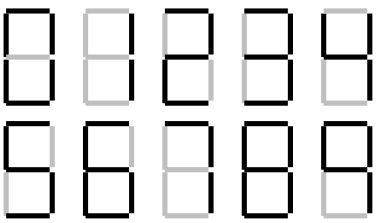
Design several two, three, and four variable truth tables by hand. Manually determine their logic equations in sum of minterms canonical form. Input the truth tables and verify the correctness of your logic equations. Include your hand designed truth tables and logic equations as well as the machine produced versions in your lab report.

6. Note: On initial entry to the program, TRUTHTBL does not display a logic equation. Therefore, you will need to set at least one square to one and then back to zero to see this equation.

Consider the following layout for a seven-segment display:



Here are the segments to light for the binary values DCBA = 0000 - 1001:



```
E = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A
F = D'C'B'A'+ D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
G = D'C'B'A' + D'C'B'A + D'C'BA' + D'C'BA + D'CB'A' + D'CBA + DC'B'A' + DC'B'A
H = D'C'BA' + D'C'BA + D'CB'A' + D'CB'A + D'CBA' + DC'B'A' + DC'B'A
I = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'
J = D'C'B'A' + D'C'B'A + D'C'BA + D'CB'A' + D'CB'A +D'CBA' + D'CBA + DC'B'A' + DC'B'A
K = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + DC'B'A'
```

Convert each of these logic equations to a truth table by setting all entries in the table to zero and then clicking on each square corresponding to each minterm in the equation. Verify by observing the equation that TRUTHTBL produces that you've successfully converted each equation to a truth table. Describe the results and provide the truth tables in your lab report.

For Additional Credit: Modify the equations above to include the following hexadecimal characters. Determine the new truth tables and use the TRUTHTBL program to verify that your truth tables and logic equations are correct.

2.9.2 Canonical Logic Equations Exercises

In this laboratory you will enter several different logic equations and compute their canonical forms as well as generate their truth table. In a sense, this exercise is the opposite of the previous exercise where you generated a canonical logic equation from a truth table.

This exercise uses the CANON.EXE program found in the CH2 subdirectory. Run this program from Windows by double clicking on its icon. This program displays a text box, a truth table, and several buttons. Unlike the TRUTHTBL.EXE program from the previous exercise, you cannot modify the truth table in the CANON.EXE program; it is a display-only table. In this program you will enter logic equations in the text entry box and then press the "Compute" button to see the resulting truth table. This program also produces the sum of minterms canonical form for the logic equation you enter (hence this program's name).

Valid logic equations take the following form:

- A *term* is either a variable (A, B, C, or D) or a logic expression surrounded by parentheses.

- A *factor* is either a term, or a factor followed by the prime symbol (an apostrophe, i.e., "'"). The prime symbol logically negates the factor immediately preceding it.
- A *product* is either a factor, or a factor concatenated with a product. The concatenation denotes logical AND operation.
- An expression is either a product or a product followed by a "+" (denoting logical OR) and followed by another expression.

Note that logical OR has the lowest precedence, logical AND has an intermediate precedence, and logical NOT has the highest precedence of these three operators. You can use parentheses to override operator precedence. The logical NOT operator, since its precedence is so high, applies only to a variable or a parenthesized expression. The following are all examples of legal expressions:

```
AB'C + D(B'+C')
AB(C+D)' + A'B'(C+D)
A'B'C'D' + ABCD + A(B+C)
(A+B)' + A'B'
```

For this set of exercises, you should create several logic expression and feed them through CANON.EXE. Include the truth tables and canonical logic forms in your lab report. Also verify that the theorems appearing in this chapter (See "Boolean Algebra" on page 43.) are valid by entering each side of the theorem and verifying that they both produce the same truth table (e.g., (AB)' = A' + B'). For additional credit, create several complex logic equations and generate their truth tables and canonical forms by hand. Then input them into the CANON.EXE program to verify your work.

2.9.3 Optimization Exercises

In this set of laboratory exercises, the OPTIMZP.EXE program (found in the CH2 subdirectory) will guide you through the steps of logic function optimization. The OPTIMZP.EXE program uses the Karnaugh Map technique to produce an equation with the minimal number of terms.

Run the OPTIMZP.EXE program by clicking on its icon or running the OPTIMZP.EXE program using the program manager's File|Run menu option. This program lets you enter an arbitrary logic equation using the same syntax as the CANON.EXE program in the previous exercise.

After entering an equation press the "Optimize" button in the OPTIMZP.EXE window. This will construct the truth table, canonical equation, and an optimized form of the logic equation you enter. Once you have optimized the equation, OPTIMZP.EXE enables the "Step" button. Pressing this button walks you through the optimization process step-by-step.

For this exercise you should enter the seven equations for the seven-segment display. Generate and record the optimize versions of these equations for your lab report and the next set of exercises. Single step through each of the equations to make sure you understand how OPTIMZP.EXE produces the optimal expressions.

For additional credit: OPTIMZP.EXE generates a single optimal expression for any given logic function. Other optimal functions may exist. Using the Karnaugh mapping technique, see if you can determine if other, equivalent, optimal expressions exist. Feed the optimal equations OPTIMZP.EXE produces and your optimal expressions into the CANON.EXE program to verify that their canonical forms are identical (and, hence, the functions are equivalent.

2.9.4 Logic Evaluation Exercises

In this set of laboratory exercises you will use the LOGIC.EXE program to enter, edit, initialize, and evaluation logic expressions. This program lets you enter up to 22 distinct

logic equations involving as many as 26 variables plus a clock value. LOGIC.EXE provides four input variables and 11 output variables (four simulated LEDs and a simulated seven-segment display). Note: this program requires that you install two files in your WINDOWS\SYSTEM directory. Please see the README.TXT file in the CH2 subdirectory for more details.

Execute the LOGIC.EXE program by double-clicking on its icon or using the program manager's "File | Run" menu option. This program consists of three main parts: an equation editor, an initialization screen, and an execution module. LOGIC.EVE uses a set of *tabbed notebook screens* to switch between these three modules. By clicking on the "Create", *Initialize*, and *Execute* tabs at the top of the screen with your mouse, you can select the specific module you want to use. Typically, you would first create a set of equations on the *Create* page and then execute those functions on the *Execute* page. Optionally, you can initialize any necessary logic variables (D-Z) on the *Initialize* page. At any time you can easily switch between modules by pressing on the appropriate notebook tab. For example, you could create a set of equations, execute them, and then go back and modify the equations (e.g., to correct any mistakes) by pressing on the *Create* tab.

The Create page lets you add, edit, and delete logic equations. Logic equations may use the variables A-Z plus the "#" symbol ("#" denotes the clock). The equations use a syntax that is very similar to the logic expressions you've used in previous exercises in this chapter. In fact, there are only two major differences between the functions LOGIC.EXE allows and the functions that the other programs allow. First, LOGIC.EXE lets you use the variables A-Z and "#" (the other programs only let you enter functions of four variables using A-D). The second difference is that LOGIC.EXE functions must take the form:

variable = expression

where *variable* is a single alphabetic character E-Z[7] and *expression* is a logic expression using the variables A-Z and #. An expression may use a maximum of four different variables (A-Z) plus the clock value (#). During the expression evaluation, the LOGIC.EXE program will evaluate the expression and store the result into the specified destination variable.

If you enter more than four variables, LOGIC.EXE will complain about your expression. LOGIC.EXE can only evaluation expressions that contain a maximum of four alphabetic characters (not counting the variable to the left of the equals sign). Note that the destination variable may appear within the expression; the following is perfectly legal:

```
F = FA+FB
```

This expression would use the *current* value of F, along with the current values of A and B to compute the new value for F.

Unlike a programming language like "C++", LOGIC.EXE does not evaluate this expression only once and store the result into F. *It will evaluate the expression several times until the value for F stabilizes.* That is, it will evaluate the expression several times until the evaluation produces the same result twice in a row. Certain expressions will produce an *infinite loop* since they will never produce the same value twice in a row. For example, the following function is unstable:

```
F = F'
```

Note that instabilities can cross function boundaries. Consider the following pair of equations:

```
F = G
G = F'
```

LOGIC.EXE will attempt to execute this set of equations until the values for the variables stop changing. However, the system above will produce an infinite loop.

---

7. A-D are read-only values that you read from a set of switches. Therefore, you cannot store a value into these variables.

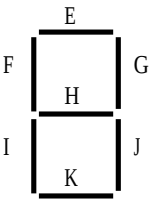
Sometimes a system of logic equations will only produce an infinite loop given certain data values. For example, consider the following of logic equation:

```
F = GF' + G'F (F = G xor F)
```

If G's value is one, this system is unstable. If G's value is zero, this equation is stable. Unstable equations like this one are somewhat harder to discover.

LOGIC.EXE will detect and warn you about logic system instabilities when you attempt to execute the logic functions. Unfortunately, it will not pinpoint the problem for you; it will simply tell you that the problem exists and expect you to fix it.

The A-D, E-K, and W-Z variables are special. A-D are read-only input variables. E-K correspond to the seven segments of a simulated seven-segment display on the *Execute* page:



W-Z correspond to four output LEDs on the *Execute* page. If the variables E-K or W-Z contain a one, then the corresponding LED (or segment) turns red (on). If the variable contains zero, the corresponding LED is off.

The *Create* page contains three important buttons: *Add, Edit*, and *Delete*. When you press the *Add* button LOGIC.EXE opens a dialog box that lets you enter an equation. Type your equation (or edit the default equation) and press the *Okay* button. If there is a problem with the equation you enter, LOGIC.EXE will report the error and make you fix the problem, otherwise, LOGIC.EXE will attempt to add this equation to the system you are building. If a function already exists that has the same destination variable as the equation you've just added, LOGIC.EXE will ask you if you really want to replace that function before proceeding with the replacement. Once LOGIC.EXE adds your equation to its list, it also displays the truth table for that equation. You can add up to 22 equations to the system (since there are 22 possible destination variables, E-Z). LOGIC.EXE displays those functions in the list box on the right hand side of the window.

Once you've entered two or more logic functions, you can view the truth table for a given logic function by simply clicking on that function with the mouse in the function list box.

If you make a mistake in a logic function you can delete that function by selecting with the mouse and pressing the *delete* button, or you can edit it by selecting it with the mouse and pressing the *edit* button. You can also edit a function by double-clicking on the function in the expression list.

The *Initialize* page displays boxes for each of the 26 possible variables. It lets you view the current values for these 26 variables and change the values of the E-Z variables (remember, A-D are read-only). As a general rule, you will not need to initialize any of the variables, so you can skip this page if you don't need to initialize any variables.

The *Execute* page contains five buttons of importance: *A-D* and *Pulse*.. The *A-D* toggle switches let you set the input values for the A-D variables. The *Pulse* switch toggles the clock value from zero to one and then back to zero, evaluating the system of logic functions while the clock is in each state.

In addition to the input buttons, there are several outputs on the *Execute* page. First, of course, are the four LEDs (W, X, Y, and Z) as well as the seven-segment display (output variables E-K as noted above). In addition to the LEDs, there is an *Instability* annunciator that turns red if LOGIC.EXE detects an instability in the system. There is also a small panel that displays the current values of all the system variables at the bottom of the window.

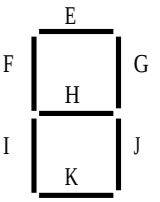
To execute the system of equations simply change one of the input values (A-D) or press the *Pulse* button. LOGIC.EXE will automatically reevaluate the system of equations whenever A-D or # changes.

To familiarize yourself with the LOGIC.EXE program, enter the following equations into the equation editor:

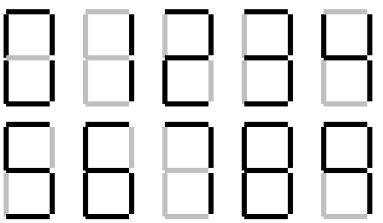
```
W = AB                  A and B
X = A + B               A or B
Y = A'B + AB'           A xor B
Z = A'                  not A
```

After entering these equations, go to the execute page and enter the four values 00, 01, 10, and 11 for BA. Note the values for W, X, Y, and Z for your lab report.

The LOGIC.EXE program simulates a seven segment display. Variables E-K light the individual segments as follows:

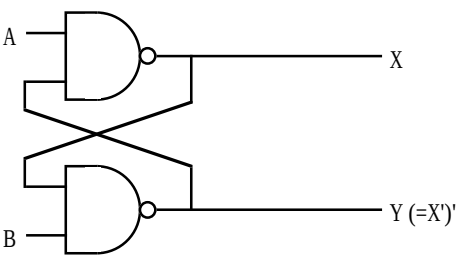


Here are the segments to light for the binary values DCBA = 0000 - 1001:



Enter the seven equations for these segments into LOGIC.EXE and try out each of the patterns (0000 through 1111). **Hint:** use the optimized equations you developed earlier. **Optional, for additional credit:** enter the equations for the 16 hexadecimal values and cycle through those 16 values. Include the results in your lab manual.

A simple sequential circuit. For this exercise you will enter the logic equations for a simple set / reset flip-flop. The circuit diagram is



A Set/Reset Flip-Flop

Since there are two outputs, this circuit has two corresponding logic equations. They are

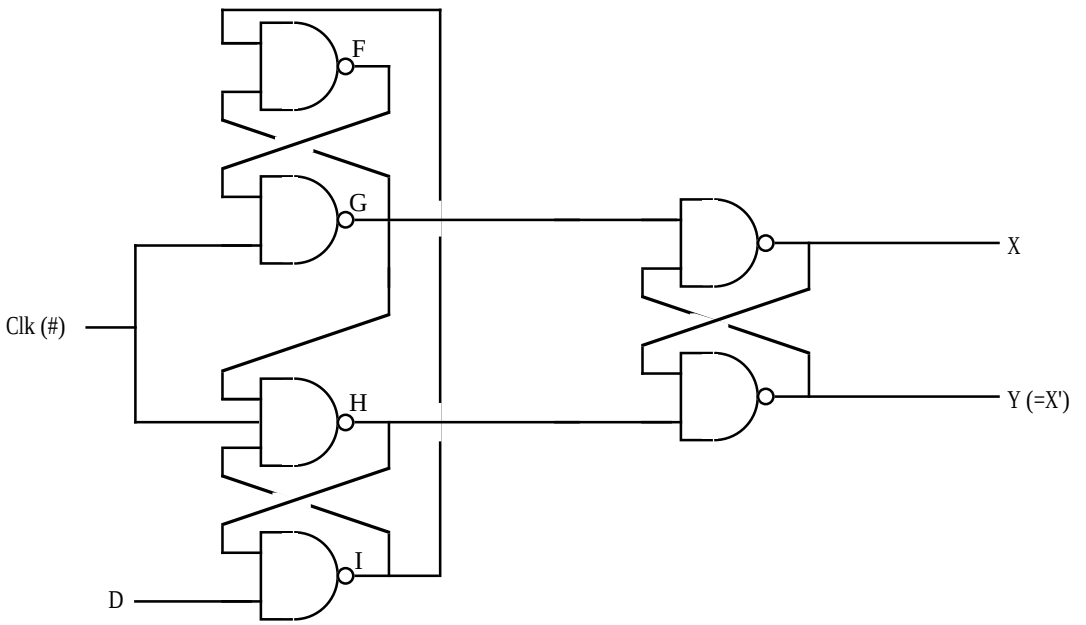
```
X = (AY)'
Y = (BX)'
```

These two equations form a *sequential circuit* since they both use variables that are function outputs. In particular, Y uses the previous value for X and X uses the previous value for Y when computing new values for X and Y.

Enter these two equations into LOGIC.EXE. Set the A and B inputs to one (the normal or *quiescent* state) and run the logic simulation. Try setting the A switch to zero and deter-

mine what happens. Press the *Pulse* button several times with A still at zero to see what happens. Then switch A back to one and repeat this process. Now try this experiment again, this time setting B to zero. Finally, try setting *both* A and B to zero and then press the *Pulse* key several times while they are zero. Then set A back to one. Try setting both to zero and then set B back to one. **For your lab report:** provide diagrams for the switch settings and resultant LED values for each time you toggle one of the buttons.

A true D flip-flop only latches the data on the D input during a clock transition from low to high. In this exercise you will simulate a D flip-flop. The circuit diagram for a true D flip-flop is

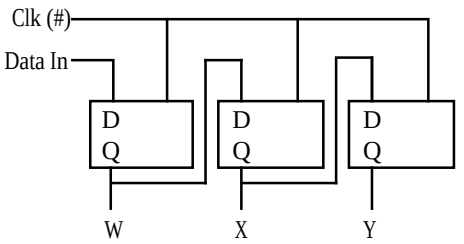


A True D flip-flop

```
F = (IG)'
G = (#F)'
H = (G#I)'
I = (DH)'
X = (GY)'
Y = (HX)'
```

Enter this set of equations and then test your flip-flop by entering different values on the D input switch and pressing the clock pulse button. Explain your results in your lab report.

In this exercise you will build a three-bit shift register using the logic equations for a true D flip-flop. To construct a shift register, you connect the outputs from each flip-flop to the input of the next flip-flop. The data input line provides the input to the first flip-flop, the last output line is the “carry out” of the circuit. Using a simple rectangle to represent a flip-flop and ignoring the Q’ output (since we don’t use it), the schematic for a four-bit shift register looks something like the following:



A Three-bit Shift Register Built from D Flip-flops

In the previous exercise you used six boolean expressions to define the D flip-flop. Therefore, we will need a total of 18 boolean expressions to implement a three-bit flip-flop. These expressions are

```
Flip-Flop #1:

        W = (GR)'
        F = (IG)'
        G = (F#)'
        H = (G#I)'
        I = (DH)'
        R = (HW)'

Flip-Flop #2:

        X = (KS)'
        J = (MK)'
        K = (J#)'
        L = (K#M)'
        M = (WL)'
        S = (LX)'

Flip-Flop #3:

        Y = (OT)'
        N = (QO)'
        O = (N#)'
        P = (O#Q)'
        Q = (XP)'
        T = (PY)'
```

Enter these equations into LOGIC.EXE. Initialize W, X, and Y to zero. Set D to one and press the *Pulse* button once to shift a one into W. Now set D to zero and press the pulse button several times to shift that single bit through each of the output bits. **For your lab report**: try shifting several bit patterns through the shift register. Describe the step-by-step operation in your lab report.

For additional credit: Describe how to create a *recirculating shift register.* One whose output from bit four feeds back into bit zero. What would be the logic equations for such a shift register? How could you initialize it (since you cannot use the D input) when using LOGIC.EXE?

Post-lab, for additional credit: Design a two-bit full adder that computes the sum of BA and DC and stores the binary result to the WXY LEDs. Include the equations and sample results in your lab report.

2.10 Programming Projects

You may write these programs in any HLL your instructor allows (typically C, C++, or some version of Borland Pascal or Delphi). You may use the generic logic functions appearing in this chapter if you so desire.

1) Write a program that reads four values from the user, I, J, K, and L, and plugs these values into a truth table with B'A' = I, B'A = J, BA' = K, and BA = L. Ensure that these input values are only zero or one. Then input a series of pairs of zeros or ones from the user and

plug them into the truth table. Display the result for each computation. Note: do *not* use the generic logic function for this program.

2) Write a program that, given a 4-bit logic function number, displays the truth table for that function of two variables.

3) Write a program that, given an 8-bit logic function number, displays the truth table for that function of three variables.

4) Write a program that, given a 16-bit logic function number, displays the truth table for that function of four variables.

5) Write a program that, given a 16-bit logic function number, displays the canonical equation for that function (hint: build the truth table).

2.11 Summary

Boolean algebra provides the foundation for both computer hardware and software. A cursory understanding of this mathematical system can help you better appreciate the connection between software and hardware.

Boolean algebra is a mathematical system with its own set of rules (postulates), theorems, and values. In many respects, boolean algebra is similar to the real-arithmetic algebra you studied in high school. In most respects, however, boolean algebra is actually *easier* to learn than real arithmetic algebra. This chapter begins by discussing features of any algebraic system including operators, closure, commutativity, associativity, distribution, identity, and inverse. Then it presents some important postulates and theorems from boolean algebra and discusses the *principle of duality* that lets you easily prove additional theorems in boolean algebra. For the details, see

- "Boolean Algebra" on page 43

The *Truth Table* is a convenient way to visually represent a boolean function or expression. Every boolean function (or expression) has a corresponding truth table that provides all possible results for any combination of input values. This chapter presents several different ways to construct boolean truth tables.

Although there are an infinite number of boolean functions you can create given *n* input values, it turns out that there are a finite number of unique functions possible for a given number of inputs. In particular, there are $2\hat{}2^n$ unique boolean functions of *n* inputs. For example, there are 16 functions of two variables ($2\hat{}2^2 = 16$).

Since there are so few boolean functions with only two inputs, it is easy to assign different names to each of these functions (e.g., AND, OR, NAND, etc.). For functions of three or more variables, the number of functions is too large to give each function its own name. Therefore, we'll assign a number to these functions based on the bits appearing in the function's truth table. For the details, see

- "Boolean Functions and Truth Tables" on page 45

We can manipulate boolean functions and expression algebraically. This allows us to prove new theorems in boolean algebra, simplify expressions, convert expressions to canonical form, or show that two expressions are equivalent. To see some examples of algebraic manipulation of boolean expressions, check out

- "Algebraic Manipulation of Boolean Expressions" on page 48

Since there are an infinite variety of possible boolean functions, yet a finite number of unique boolean functions (for a fixed number of inputs), clearly there are an infinite number of different functions that compute the same results. To avoid confusion, logic designers usually specify a boolean function using a *canonical form*. If two canonical equations are different, then they represent different boolean functions. This book describes two different canonical forms: the sum of minterms form and the product of maxterms form. To

learn about these canonical forms, how to convert an arbitrary boolean equation to canonical form, and how to convert between the two canonical forms, see

- “Canonical Forms” on page 49

Although the canonical forms provide a unique representation for a given boolean function, expressions appearing in canonical form are rarely *optimal*. That is, canonical expressions often use more literals and operators than other, equivalent, expressions. When designing an electronic circuit or a section of software involving boolean expressions, most engineers prefer to use an optimized circuit or program since optimized versions are less expensive and, probably, faster. Therefore, knowing how to create an optimized form of a boolean expression is very important. This text discusses this subject in

- “Simplification of Boolean Functions” on page 52

Boolean algebra isn’t a system designed by some crazy mathematician that has little importance in the real world. Boolean algebra is the basis for digital logic that is, in turn, the basis for computer design. Furthermore, there is a one-to-one correspondence between digital hardware and computer software. Anything you can build in hardware you can construct with software, and vice versa. This text describes how to implement addition, decoders, memory, shift registers, and counters using these boolean functions. Likewise, this text describes how to improve the efficiency of software (e.g., a Pascal program) by applying the rules and theorems of boolean algebra. For all the details, see

- “What Does This Have To Do With Computers, Anyway?” on page 59
- “Correspondence Between Electronic Circuits and Boolean Functions” on page 59
- “Combinatorial Circuits” on page 60
- “Sequential and Clocked Logic” on page 62
- “Okay, What Does It Have To Do With Programming, Then?” on page 64

2.12 Questions

1. What is the identity element with respect to

   a) AND b) OR c) XOR d) NOT e) NAND f) NOR

2. Provide truth tables for the following functions of two input variables:

   a) AND b) OR c) XOR d) NAND e) NOR

   f) Equivalence g) A < B h) A > B i) A implies B

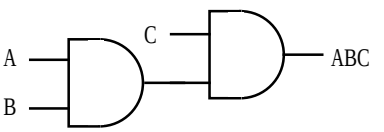
3. Provide the truth tables for the following functions of three input variables:

   a) ABC (and) b) A+B+C (OR) c) (ABC)' (NAND)d) (A+B+C)' (NOR)

   e) Equivalence (ABC) + (A'B'C') f) XOR (ABC + A'B'C')'

4. Provide schematics (electrical circuit diagrams) showing how to implement each of the functions in question three using only two-input gates and inverters. E.g.,

   A) ABC =



5. Provide implementations of an AND gate, OR gate, and inverter gate using one or more NOR gates.

6. What is the principle of duality? What does it do for us?

7. Build a single truth table that provides the outputs for the following three boolean functions of three variables:

   F_x= A + BC

   F_y - AB + C'B

   F_z = A'B'C' + ABC + C'B'A

8. Provide the function numbers for the three functions in question seven above.

9. How many possible (unique) boolean functions are there if the function has

   a) one input b) two inputs c) three inputs d) four inputs e) five inputs

10. Simplify the following boolean functions using algebraic transformations. Show your work.

    a) F = AB + AB' b) F = ABC + BC' + AC + ABC'

    c) F = A'B'C'D' + A'B'C'D + A'B'CD + A'B'CD'

    d) F = A'BC + ABC' + A'BC' + AB'C' + ABC + AB'C

11. Simplify the boolean functions in question 10 using the mapping method.

12. Provide the logic equations in canonical form for the boolean functions S_0..S_6 for the seven segment display (see "Combinatorial Circuits" on page 60).

13. Provide the truth tables for each of the functions in question 12

14. Minimize each of the functions in question 12 using the map method.

15. The logic equation for a half-adder (in canonical form) is

    Sum = AB' + A'B Carry = AB

    a) Provide an electronic circuit diagram for a half-adder using AND, OR, and Inverter gates

    b) Provide the circuit using only NAND gates

16. The canonical equations for a full adder take the form:

    Sum = A'B'C + A'BC' + AB'C' + ABC

    Carry = ABC + ABC' + AB'C + A'BC

    a) Provide the schematic for these circuits using AND, OR, and inverter gates.

    b) Optimize these equations using the map method.

    c) Provide the electronic circuit for the optimized version (using AND, OR, and inverter gates).

17. Assume you have a D flip-flop (use this definition in this text) whose outputs currently are Q=1 and Q'=0. Describe, in minute detail, exactly what happens when the clock line goes

    a) from low to high with D=0

    b) from high to low with D=0

18. Rewrite the following Pascal statements to make them more efficient:

    a) if (x or (not x and y)) then write('1');

    b) while(not x and not y) do somefunc(x,y);

    c) if not((x <> y) and (a = b)) then Something;

19. Provide canonical forms (sum of minterms) for each of the following:

    a) F(A,B,C) = A'BC + AB + BC b) F(A,B,C,D) = A + B + CD' + D

    c) F(A,B,C) = A'B + B'A d) F(A,B,C,D) = A + BD'

    e) F(A,B,C,D) = A'B'C'D + AB'C'D' + CD + A'BCD'

20. Convert the sum of minterms forms in question 19 to the product of maxterms forms.

To write even a modest 80x86 assembly language program requires considerable familiarity with the 80x86 family. To write *good* assembly language programs requires a strong knowledge of the underlying hardware. Unfortunately, the underlying hardware is not consistent. Techniques that are crucial for 8088 programs may not be useful on 80486 systems. Likewise, programming techniques that provide big performance boosts on an 80486 chip may not help at all on an 80286. Fortunately, some programming techniques work well whatever microprocessor you're using. This chapter discusses the effect hardware has on the performance of computer software.

3.0 Chapter Overview

This chapter describes the basic components that make up a computer system: the CPU, memory, I/O, and the bus that connects them. Although you can write software that is ignorant of these concepts, high performance software requires a complete understanding of this material.

This chapter begins by discussing bus organization and memory organization. These two hardware components will probably have a bigger performance impact on your software than the CPU's speed. Understanding the organization of the system bus will allow you to design data structures that operate and maximum speed. Similarly, knowing about memory performance characteristics, data locality, and cache operation can help you design software that runs as fast as possible. Of course, if you're not interested in writing code that runs as fast as possible, you can skip this discussion; however, most people do care about speed at one point or another, so learning this information is useful.

Unfortunately, the 80x86 family microprocessors are a complex group and often overwhelm beginning students. Therefore, this chapter describes four hypothetical members of the 80x86 family: the 886, 8286, the 8486, and the 8686 microprocessors. These represent simplified versions of the 80x86 chips and allow a discussion of various architectural features without getting bogged down by huge CISC instruction sets. This text uses the x86 hypothetical processors to describe the concepts of instruction encoding, addressing modes, sequential execution, the prefetch queue, pipelining, and superscalar operation. Once again, these are concepts you do not need to learn if you only want to write *correct* software. However, if you want to write *fast* software as well, especially on advanced processors like the 80486, Pentium, and beyond, you will need to learn about these concepts.

Some might argue that this chapter gets too involved with computer architecture. They feel such material should appear in an architectural book, not an assembly language programming book. This couldn't be farther from the truth! Writing *good* assembly language programs requires a strong knowledge of the architecture. Hence the emphasis on computer architecture in this chapter.

3.1 The Basic System Components

The basic operational design of a computer system is called its *architecture*. John Von Neumann, a pioneer in computer design, is given credit for the architecture of most computers in use today. For example, the 80x86 family uses the *Von Neumann architecture* (VNA). A typical Von Neumann system has three major components: the *central processing unit* (or *CPU*), *memory*, and *input/output* (or *I/O*). The way a system designer combines these components impacts system performance (see Figure 3.1).

In VNA machines, like the 80x86 family, the CPU is where all the action takes place. All computations occur inside the CPU. Data and CPU instructions reside in memory until required by the CPU. To the CPU, most I/O devices look like memory because the

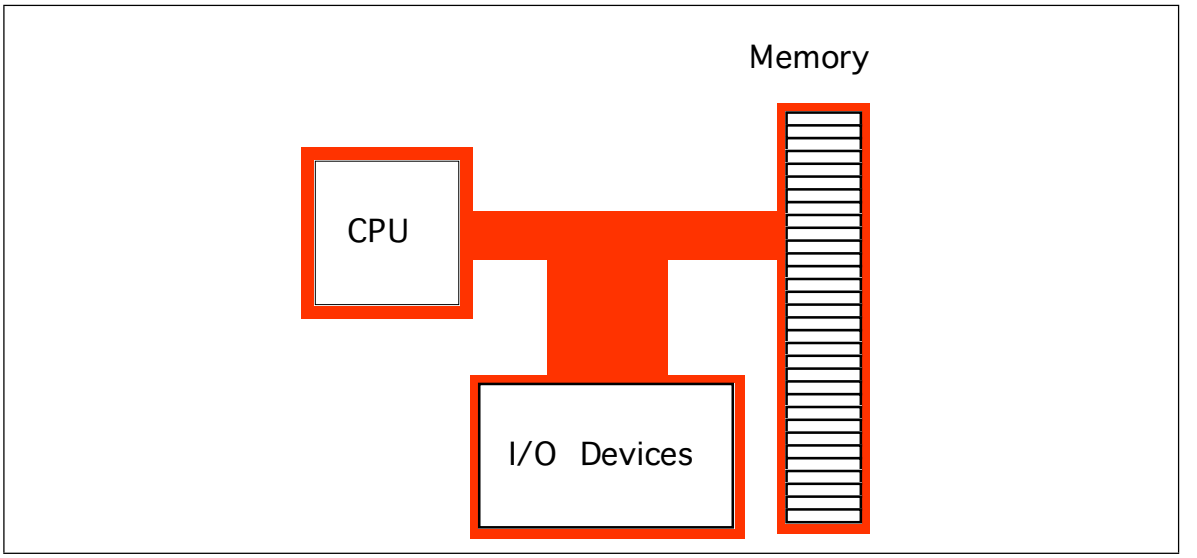


Figure 3.1 Typical Von Neumann Machine

CPU can store data to an output device and read data from an input device. The major difference between memory and I/O locations is the fact that I/O locations are generally associated with external devices in the outside world.

3.1.1 The System Bus

The *system bus* connects the various components of a VNA machine. The 80x86 family has three major busses: the *address* bus, the *data* bus, and the *control* bus. A bus is a collection of wires on which electrical signals pass between components in the system. These busses vary from processor to processor. However, each bus carries comparable information on all processors; e.g., the data bus may have a different implementation on the 80386 than on the 8088, but both carry data between the processor, I/O, and memory.

A typical 80x86 system component uses *standard TTL logic levels.* This means each wire on a bus uses a standard voltage level to represent zero and one[1]. We will always specify zero and one rather than the electrical levels because these levels vary on different processors (especially laptops).

3.1.1.1 The Data Bus

The 80x86 processors use the *data bus* to shuffle data between the various components in a computer system. The size of this bus varies widely in the 80x86 family. Indeed, this bus defines the "size" of the processor.

On typical 80x86 systems, the data bus contains eight, 16, 32, or 64 lines. The 8088 and 80188 microprocessors have an eight bit data bus (eight data lines). The 8086, 80186, 80286, and 80386SX processors have a 16 bit data bus. The 80386DX, 80486, and Pentium Overdrive™ processors have a 32 bit data bus. The Pentium™ and Pentium Pro processors have a 64 bit data bus. Future versions of the chip (the 80686/80786?) may have a larger bus.

Having an eight bit data bus does not limit the processor to eight bit data types. It simply means that the processor can only access one byte of data per memory cycle (see

1. TTL logic represents the value zero with a voltage in the range 0.0-0.8v. It represents a one with a voltage in the range 2.4-5v. If the signal on a bus line is between 0.8v and 2.4v, it's value is indeterminate. Such a condition should only exist when a bus line is changing from one state to the other.

The "Size" of a Processor

There has been a considerable amount of disagreement among hardware and software engineers concerning the "size" of a processor like the 8088. From a hardware designer's perspective, the 8088 is purely an eight bit processor – it has only eight data lines and is bus compatible with memory and I/O devices designed for eight bit processors. Software engineers, on the other hand, have argued that the 8088 is a 16 bit processor. From their perspective they cannot distinguish between the 8088 (with an eight-bit data bus) and the 8086 (which has a 16-bit data bus). Indeed, the only difference is the speed at which the two processors operate; the 8086 with a 16 bit data bus is faster. Eventually, the hardware designers won out. Despite the fact that software engineers cannot differentiate the 8088 and 8086 in their programs, we call the 8088 an eight bit processor and the 8086 a 16 bit processor. Likewise, the 80386SX (which has a sixteen bit data bus) is a 16 bit processor while the 80386DX (which has a full 32 bit data bus) is a 32 bit processor.

"The Memory Subsystem" on page 87 for a description of memory cycles). Therefore, the eight bit bus on an 8088 can only transmit half the information per unit time (memory cycle) as the 16 bit bus on the 8086. Therefore, processors with a 16 bit bus are naturally faster than processors with an eight bit bus. Likewise, processors with a 32 bit bus are faster than those with a 16 or eight bit data bus. The size of the data bus affects the performance of the system more than the size of any other bus.

You'll often hear a processor called an *eight, 16, 32, or 64 bit processor.* While there is a mild controversy concerning the size of a processor, most people now agree that the number of data lines on the processor determines its size. Since the 80x86 family busses are eight, 16, 32, or 64 bits wide, most data accesses are also eight, 16, 32, or 64 bits. Although it is possible to process 12 bit data with an 8088, most programmers process 16 bits since the processor will fetch and manipulate 16 bits anyway. This is because the processor always fetches eight bits. To fetch 12 bits requires two eight bit memory operations. Since the processor fetches 16 bits rather than 12, most programmers use all 16 bits. In general, manipulating data which is eight, 16, 32, or 64 bits in length is the most efficient.

Although the 16, 32, and 64 bit members of the 80x86 family *can* process data up to the width of the bus, they can also access smaller memory units of eight, 16, or 32 bits. Therefore, anything you can do with a small data bus can be done with a larger data bus as well; the larger data bus, however, may access memory faster and can access larger chunks of data in one memory operation. You'll read about the exact nature of these memory accesses a little later (see "The Memory Subsystem" on page 87).

Table 17: 80x86 Processor Data Bus Sizes

Processor	Data Bus Size
8088	8
80188	8
8086	16
80186	16
80286	16
80386sx	16
80386dx	32
80486	32
80586 class/ Pentium (Pro)	64

3.1.1.2 The Address Bus

The data bus on an 80x86 family processor transfers information between a particular memory location or I/O device and the CPU. The only question is, "*Which memory location or I/O device?*" The address bus answers that question. To differentiate memory locations and I/O devices, the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device, it places the corresponding address on the address bus. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on the data bus. In either case, all other memory locations ignore the request. Only the device whose address matches the value on the address bus responds.

With a single address line, a processor could create exactly two unique addresses: zero and one. With *n* address lines, the processor can provide 2^n unique addresses (since there are 2^n unique values in an *n*-bit binary number). Therefore, the number of bits on the address bus will determine the *maximum* number of addressable memory and I/O locations. The 8088 and 8086, for example, have 20 bit address busses. Therefore, they can access up to 1,048,576 (or 2^{20}) memory locations. Larger address busses can access more memory. The 8088 and 8086, for example, suffer from an anemic address space[2] – their address bus is too small. Later processors have larger address busses:

Table 18: 80x86 Family Address Bus Sizes

Processor	Address Bus Size	Max Addressable Memory	In English!
8088	20	1,048,576	One Megabyte
8086	20	1,048,576	One Megabyte
80188	20	1,048,576	One Megabyte
80186	20	1,048,576	One Megabyte
80286	24	16,777,216	Sixteen Megabytes
80386sx	24	16,777,216	Sixteen Megabytes
80386dx	32	4,294,976,296	Four Gigabytes
80486	32	4,294,976,296	Four Gigabytes
80586 / Pentium (Pro)	32	4,294,976,296	Four Gigabytes

Future 80x86 processors will probably support 48 bit address busses. The time is coming when most programmers will consider four gigabytes of storage to be too small, much like they consider one megabyte insufficient today. (There was a time when one megabyte was considered far more than anyone would ever need!) Fortunately, the architecture of the 80386, 80486, and later chips allow for an easy expansion to a 48 bit address bus through *segmentation*.

3.1.1.3 The Control Bus

The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. Consider for a moment the data bus. The CPU sends data to memory and receives data from memory on the data bus. This prompts the question, "Is it sending or receiving?" There are two lines on the control bus, *read* and *write*, which specify the direction of data flow. Other signals include system clocks, interrupt lines, status lines, and so on. The exact make up of the control bus varies among pro-

2. The address space is the set of all addressable memory locations.

cessors in the 80x86 family. However, some control lines are common to all processors and are worth a brief mention.

The *read* and *write* control lines control the direction of data on the data bus. When both contain a logic one, the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero), the CPU is reading data from memory (that is, the system is transferring data from memory to the CPU). If the write line is low, the system transfers data from the CPU to memory.

The *byte enable lines* are another set of important control lines. These control lines allow 16, 32, and 64 bit processors to deal with smaller chunks of data. Additional details appear in the next section.

The 80x86 family, unlike many other processors, provides two distinct address spaces: one for memory and one for I/O. While the memory address busses on various 80x86 processors vary in size, the I/O address bus on all 80x86 CPUs is 16 bits wide. This allows the processor to address up to 65,536 different I/O *locations*. As it turns out, most devices (like the keyboard, printer, disk drives, etc.) require more than one I/O location. Nonetheless, 65,536 I/O locations are more than sufficient for most applications. The original IBM PC design only allowed the use of 1,024 of these.

Although the 80x86 family supports two address spaces, it does not have two address busses (for I/O and memory). Instead, the system shares the address bus for both I/O and memory addresses. Additional control lines decide whether the address is intended for memory or I/O. When such signals are active, the I/O devices use the address on the L.O. 16 bits of the address bus. When inactive, the I/O devices ignore the signals on the address bus (the memory subsystem takes over at that point).

3.1.2 The Memory Subsystem

A typical 80x86 processor addresses a maximum of 2^n different memory locations, where *n* is the number of bits on the address bus[3]. As you've seen already, 80x86 processors have 20, 24, and 32 bit address busses (with 48 bits on the way).

Of course, the first question you should ask is, "What exactly is a memory location?" The 80x86 supports *byte addressable memory*. Therefore, the basic memory unit is a byte. So with 20, 24, and 32 address lines, the 80x86 processors can address one megabyte, 16 megabytes, and four gigabytes of memory, respectively.

Think of memory as a linear array of bytes. The address of the first byte is zero and the address of the last byte is 2^n-1. For an 8088 with a 20 bit address bus, the following pseudo-Pascal array declaration is a good approximation of memory:

```
Memory: array [0..1048575] of byte;
```

To execute the equivalent of the Pascal statement "Memory [125] := 0;" the CPU places the value zero on the data bus, the address 125 on the address bus, and asserts the write line (since the CPU is writing data to memory, see Figure 3.2)

To execute the equivalent of "CPU := Memory [125];" the CPU places the address 125 on the address bus, asserts the read line (since the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 3.2).

The above discussion applies *only* when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Since memory consists of an array of bytes, how can we possibly deal with values larger than eight bits?

Different computer systems have different solutions to this problem. The 80x86 family deals with this problem by storing the L.O. byte of a word at the address specified and the H.O. byte at the next location. Therefore, a word consumes two consecutive memory

3. This is the *maximum*. Most computer systems built around 80x86 family do not include the maximum addressable amount of memory.

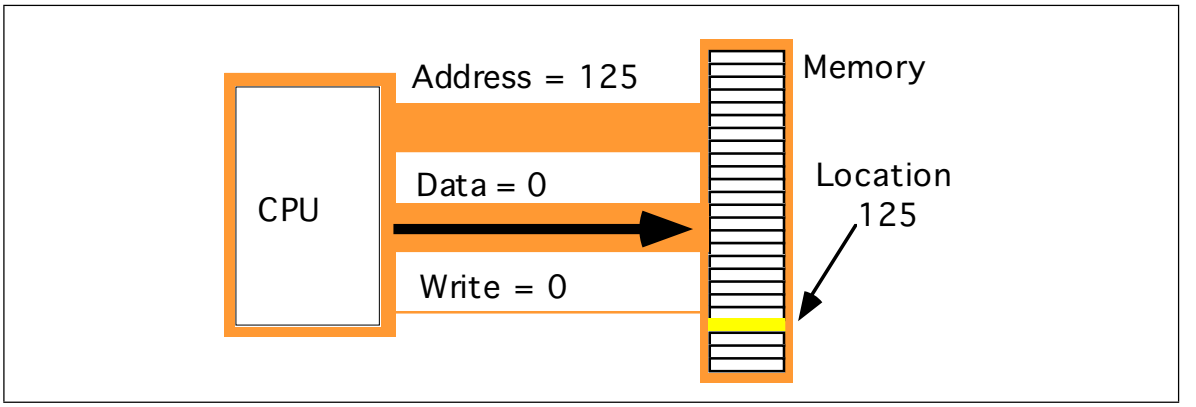


Figure 3.2 Memory Write Operation

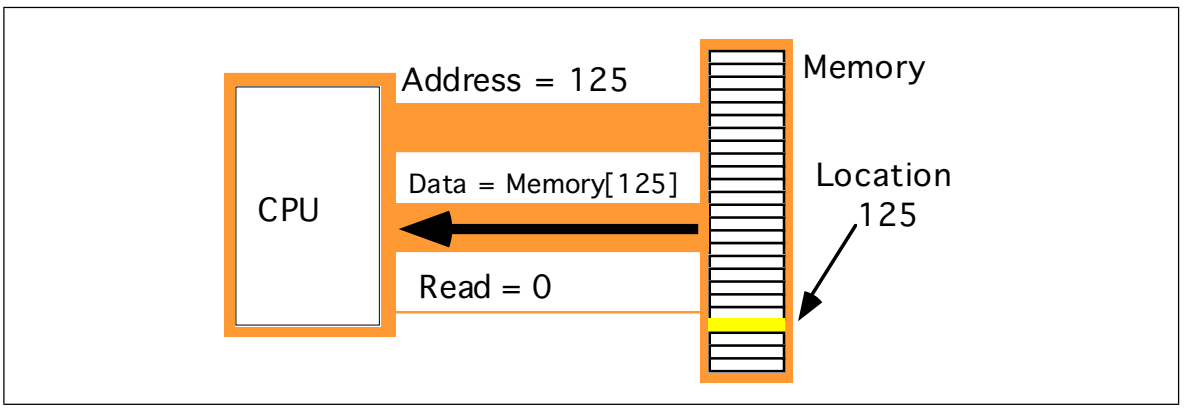


Figure 3.3 Memory Read Operation

addresses (as you would expect, since a word consists of two bytes). Similarly, a double word consumes four consecutive memory locations. The address for the double word is the address of its L.O. byte. The remaining three bytes follow this L.O. byte, with the H.O. byte appearing at the address of the double word *plus three* (see Figure 3.4) Bytes, words, and double words may begin at *any* valid address in memory. We will soon see, however, that starting larger objects at an arbitrary address is not a good idea.

Note that it is quite possible for byte, word, and double word values to overlap in memory. For example, in Figure 3.4 you could have a word variable beginning at address 193, a byte variable at address 194, and a double word value beginning at address 192. These variables would all overlap.

The 8088 and 80188 microprocessors have an eight bit data bus. This means that the CPU can transfer eight bits of data at a time. Since each memory address corresponds to an eight bit byte, this turns out to be the most convenient arrangement (from the hardware perspective), see Figure 3.5.

The term “byte addressable memory array” means that the CPU can address memory in chunks as small as a single byte. It also means that this is the *smallest* unit of memory you can access at once with the processor. That is, if the processor wants to access a four bit value, it must read eight bits and then ignore the extra four bits. Also realize that byte addressability does not imply that the CPU can access eight bits on any arbitrary bit boundary. When you specify address 125 in memory, you get the entire eight bits at that address, nothing less, nothing more. Addresses are integers; you cannot, for example, specify address 125.5 to fetch fewer than eight bits.

The 8088 and 80188 can manipulate word and double word values, even with their eight bit data bus. However, this requires multiple memory operations because these processors can only move eight bits of data at once. To load a word requires two memory operations; to load a double word requires four memory operations.

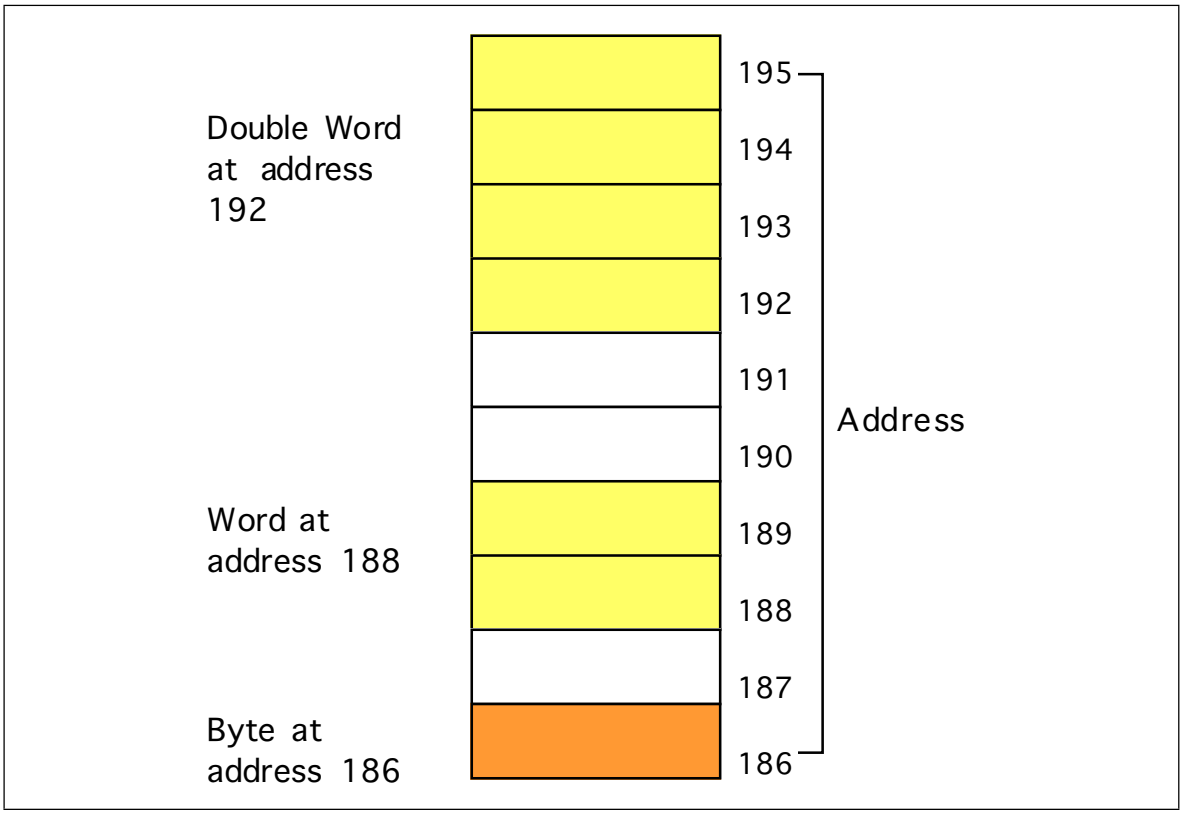


Figure 3.4 Byte, Word, and Double word Storage in Memory

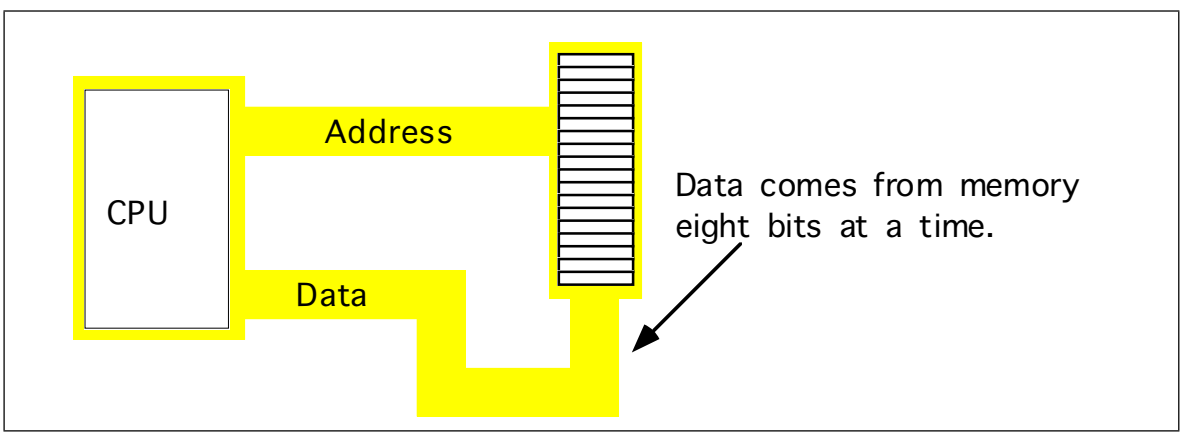


Figure 3.5 Eight-Bit CPU-Memory Interface

The 8086, 80186, 80286, and 80386sx processors have a 16 bit data bus. This allows these processors to access twice as much memory in the same amount of time as their eight bit brethren. These processors organize memory into two *banks*: an "even" bank and an "odd" bank (see Figure 3.6). Figure 3.7 illustrates the connection to the CPU (D0-D7 denotes the L.O. byte of the data bus, D8-D15 denotes the H.O. byte of the data bus):

The 16 bit members of the 80x86 family can load a word from any arbitrary address. As mentioned earlier, the processor fetches the L.O. byte of the value from the address specified and the H.O. byte from the next consecutive address. This creates a subtle problem if you look closely at the diagram above. What happens when you access a word on an odd address? Suppose you want to read a word from location 125. Okay, the L.O. byte of the word comes from location 125 and the H.O. word comes from location 126. What's the big deal? It turns out that there are two problems with this approach.

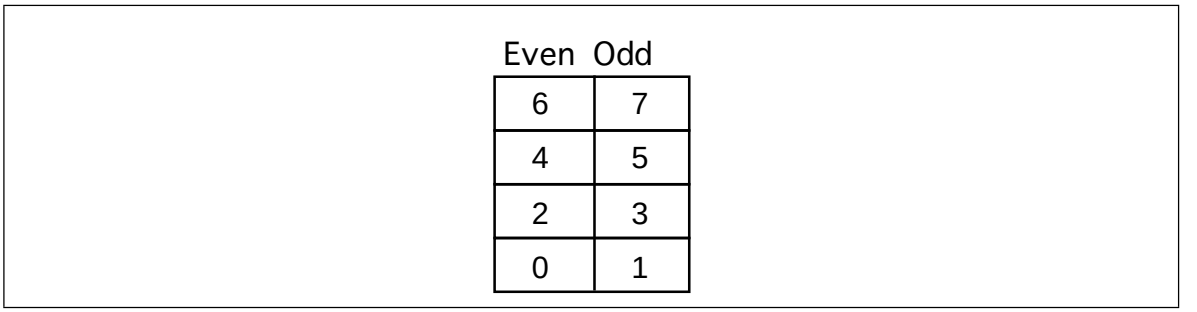


Figure 3.6 Byte Addresses in Word Memory

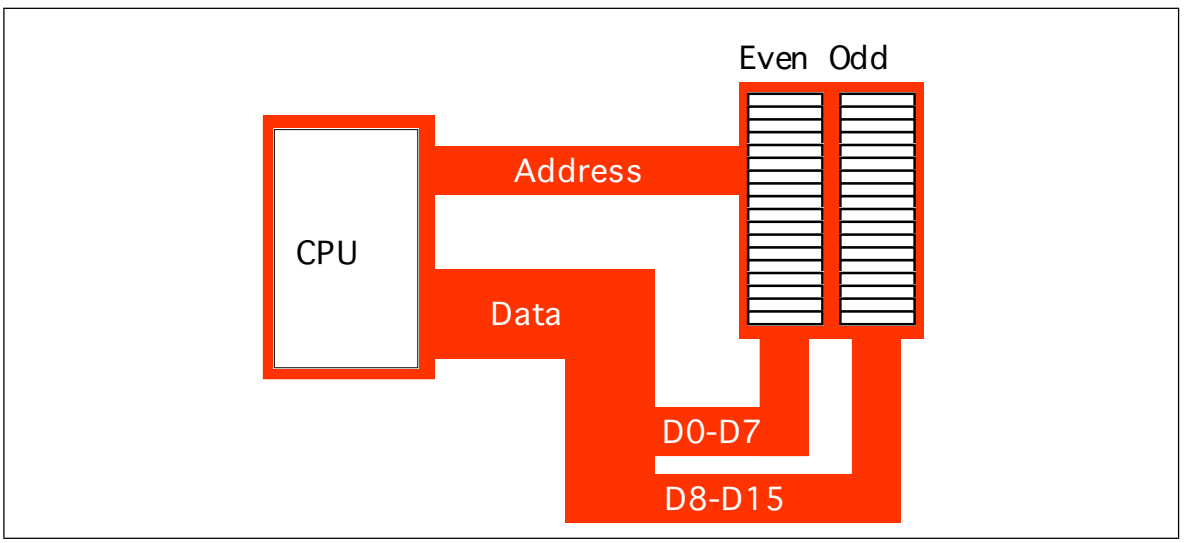


Figure 3.7 16-Bit Processor (8086, 80186, 80286, 80386sx) Memory Organization

First, look again at Figure 3.7. Data bus lines eight through 15 (the H.O. byte) connect to the odd bank, and data bus lines zero through seven (the L.O. byte) connect to the even bank. Accessing memory location 125 will transfer data to the CPU on the H.O. byte of the data bus; yet we want this data in the L.O. byte! Fortunately, the 80x86 CPUs recognize this situation and automatically transfer the data on D8-D15 to the L.O. byte.

The second problem is even more obscure. When accessing words, we're really accessing two separate bytes, each of which has its own byte address. So the question arises, "What address appears on the address bus?" The 16 bit 80x86 CPUs always place even addresses on the bus. Even bytes always appear on data lines D0-D7 and the odd bytes always appear on data lines D8-D15. If you access a word at an even address, the CPU can bring in the entire 16 bit chunk in one memory operation. Likewise, if you access a single byte, the CPU activates the appropriate bank (using a "byte enable" control line). If the byte appeared at an odd address, the CPU will automatically move it from the H.O. byte on the bus to the L.O. byte.

So what happens when the CPU accesses a *word* at an odd address, like the example given earlier? Well, the CPU cannot place the address 125 onto the address bus and read the 16 bits from memory. There are no odd addresses coming out of a 16 bit 80x86 CPU. The addresses are always even. So if you try to put 125 on the address bus, this will put 124 on to the address bus. Were you to read the 16 bits at this address, you would get the word at addresses 124 (L.O. byte) and 125 (H.O. byte) – not what you'd expect. Accessing a word at an odd address requires two memory operations. First the CPU must read the byte at address 125, then it needs to read the byte at address 126. Finally, it needs to swap the positions of these bytes internally since both entered the CPU on the wrong half of the data bus.

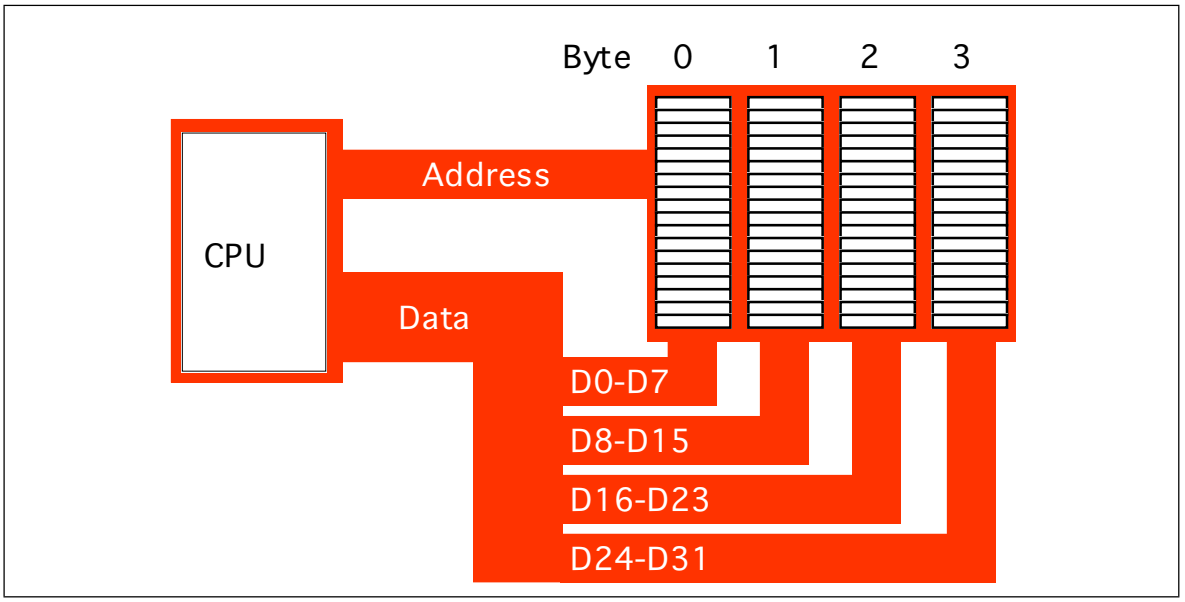


Figure 3.8 32-Bit Processor (80386, 80486, Pentium Overdrive) Memory Organization

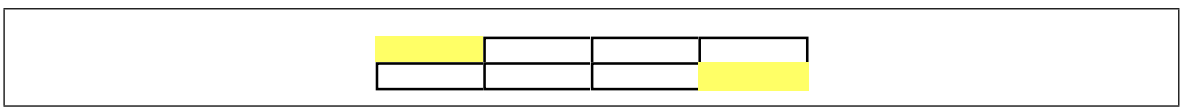


Figure 3.9 Accessing a Word at (Address **mod** 4) = 3.

Fortunately, the 16 bit 80x86 CPUs hide these details from you. Your programs can access words at *any* address and the CPU will properly access and swap (if necessary) the data in memory. However, to access a word at an odd address requires two memory operations (just like the 8088/80188). Therefore, accessing words at odd addresses on a 16 bit processor is slower than accessing words at even addresses. **By carefully arranging how you use memory, you can improve the speed of your program.**

Accessing 32 bit quantities always takes at least two memory operations on the 16 bit processors. If you access a 32 bit quantity at an odd address, the processor will require three memory operations to access the data.

The 32 bit 80x86 processors (the 80386, 80486, and Pentium Overdrive) use four banks of memory connected to the 32 bit data bus (see Figure 3.8). The address placed on the address bus is always some multiple of four. Using various “byte enable” lines, the CPU can select which of the four bytes at that address the software wants to access. As with the 16 bit processor, the CPU will automatically rearrange bytes as necessary.

With a 32 bit memory interface, the 80x86 CPU can access any byte with one memory operation. If (address MOD 4) does not equal three, then a 32 bit CPU can access a word at that address using a single memory operation. However, if the remainder is three, then it will take two memory operations to access that word (see Figure 3.9). This is the same problem encountered with the 16 bit processor, except it occurs half as often.

A 32 bit CPU can access a double word in a single memory operation *if* the address of that value is evenly divisible by four. If not, the CPU will require two memory operations.

Once again, the CPU handles all of this automatically. In terms of loading correct data the CPU handles everything for you. However, there is a performance benefit to proper data alignment. As a general rule you should always place word values at even addresses and double word values at addresses which are evenly divisible by four. This will speed up your program.

3.1.3 The I/O Subsystem

Besides the 20, 24, or 32 address lines which access memory, the 80x86 family provides a 16 bit I/O address bus. This gives the 80x86 CPUs two separate address spaces: one for memory and one for I/O operations. Lines on the control bus differentiate between memory and I/O addresses. Other than separate control lines and a smaller bus, I/O addressing behaves exactly like memory addressing. Memory and I/O devices both share the same data bus and the L.O. 16 lines on the address bus.

There are three limitations to the I/O subsystem on the IBM PC: first, the 80x86 CPUs require special instructions to access I/O devices; second, the designers of the IBM PC used the "best" I/O locations for their own purposes, forcing third party developers to use less accessible locations; third, 80x86 systems can address no more than 65,536 (2^{16}) I/O addresses. When you consider that a typical VGA display card requires over 128,000 different locations, you can see a problem with the size of I/O bus.

Fortunately, hardware designers can map their I/O devices into the memory address space as easily as they can the I/O address space. So by using the appropriate circuitry, they can make their I/O devices look just like memory. This is how, for example, display adapters on the IBM PC work.

Accessing I/O devices is a subject we'll return to in later chapters. For right now you can assume that I/O and memory accesses work the same way.

3.2 System Timing

Although modern computers are quite fast and getting faster all the time, they still require a finite amount of time to accomplish even the smallest tasks. On Von Neumann machines, like the 80x86, most operations are *serialized*. This means that the computer executes commands in a prescribed order. It wouldn't do, for example, to execute the statement I:=I*5+2; before I:=J; in the following sequence:

```
I := J;
I := I * 5 + 2;
```

Clearly we need some way to control which statement executes first and which executes second.

Of course, on real computer systems, operations do not occur instantaneously. Moving a copy of J into I takes a certain amount of time. Likewise, multiplying I by five and then adding two and storing the result back into I takes time. As you might expect, the second Pascal statement above takes quite a bit longer to execute than the first. For those interested in writing fast software, a natural question to ask is, "How does the processor execute statements, and how do we measure how long they take to execute?"

The CPU is a very complex piece of circuitry. Without going into too many details, let us just say that operations inside the CPU must be very carefully coordinated or the CPU will produce erroneous results. To ensure that all operations occur at just the right moment, the 80x86 CPUs use an alternating signal called the *system clock*.

3.2.1 The System Clock

At the most basic level, the *system clock* handles all synchronization within a computer system. The system clock is an electrical signal on the control bus which alternates between zero and one at a periodic rate (see Figure 3.10). CPUs are a good example of a complex synchronous logic system (see the previous chapter). The system clock gates many of the logic gates that make up the CPU allowing them to operate in a synchronized fashion.

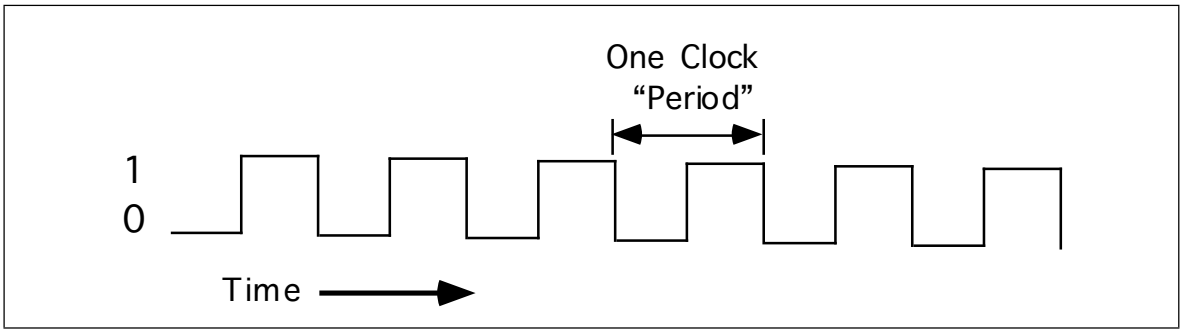


Figure 3.10 The System Clock

The frequency with which the system clock alternates between zero and one is the *system clock frequency*. The time it takes for the system clock to switch from zero to one and back to zero is the *clock period*. One full period is also called a *clock cycle*. On most modern systems, the system clock switches between zero and one at rates exceeding several million times per second. The clock frequency is simply the number of clock cycles which occur each second. A typical 80486 chip runs at speeds of 66million cycles per second. "Hertz" (Hz) is the technical term meaning one cycle per second. Therefore, the aforementioned 80486 chip runs at 66 million hertz, or 66 megahertz (MHz). Typical frequencies for 80x86 parts range from 5 MHz up to 200 MHz and beyond. Note that one clock period (the amount of time for one complete clock cycle) is the reciprocal of the clock frequency. For example, a 1 MHz clock would have a clock period of one microsecond ($1 / 1{,}000{,}000^{th}$ of a second). Likewise, a 10 MHz clock would have a clock period of 100 nanoseconds (100 billionths of a second). A CPU running at 50 MHz would have a clock period of 20 nanoseconds. Note that we usually express clock periods in millionths or billionths of a second.

To ensure synchronization, most CPUs start an operation on either the *falling edge* (when the clock goes from one to zero) or the *rising edge* (when the clock goes from zero to one). The system clock spends most of its time at either zero or one and very little time switching between the two. Therefore clock edge is the perfect synchronization point.

Since all CPU operations are synchronized around the clock, the CPU cannot perform tasks any faster than the clock[4]. However, just because a CPU is running at some clock frequency doesn't mean that it is executing that many operations each second. Many operations take multiple clock cycles to complete so the CPU often performs operations at a significantly lower rate.

3.2.2 Memory Access and the System Clock

Memory access is probably the most common CPU activity. Memory access is definitely an operation synchronized around the system clock. That is, reading a value from memory or writing a value to memory occurs no more often than once every clock cycle[5]. Indeed, on many 80x86 processors, it takes several clock cycles to access a memory location. The *memory access time* is the number of clock cycles the system requires to access a memory location; this is an important value since longer memory access times result in lower performance.

Different 80x86 processors have different memory access times ranging from one to four clock cycles. For example, the 8088 and 8086 CPUs require *four* clock cycles to access memory; the 80486 requires only one. Therefore, the 80486 will execute programs which access memory faster than an 8086, even when running at the same clock frequency.

4. Some later versions of the 80486 use special clock doubling circuitry to run twice as fast as the input clock frequency. For example, with a 25 MHz clock the chip runs at an effective rate of 50 MHz. However, the internal clock frequency *is* 50 MHz. The CPU still won't execute operations faster than 50 million operations per second.
5. This is true even on the clock doubled CPUs.

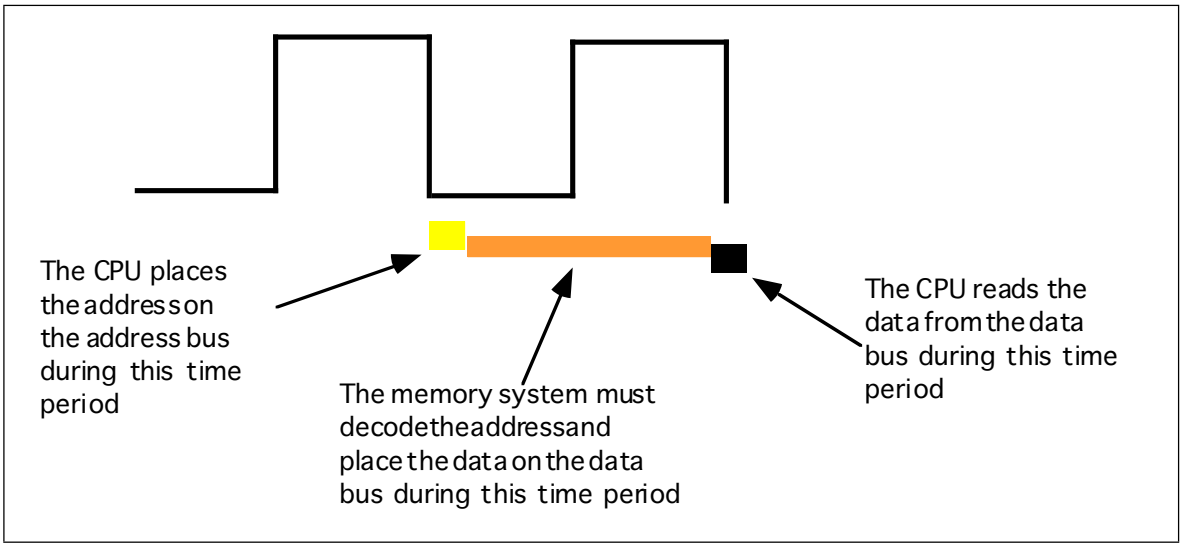


Figure 3.11 An 80486 Memory Read Cycle

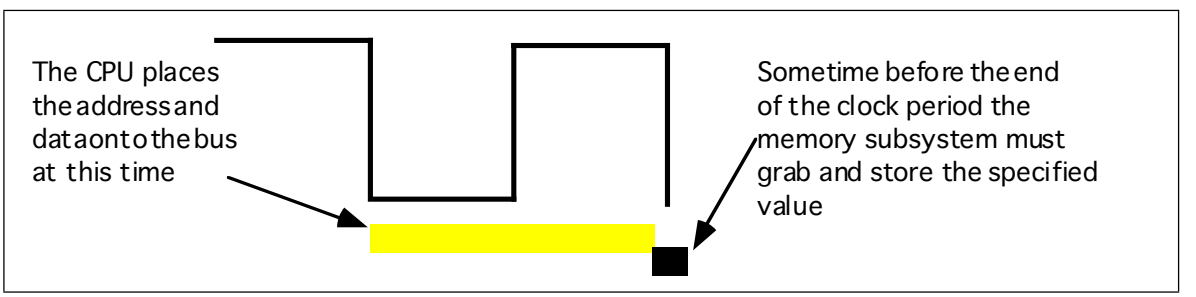


Figure 3.12 An 80486 Memory Write Cycle

Memory access time is the amount of time between a memory operation request (read or write) and the time the memory operation completes. On a 5 MHz 8088/8086 CPU the memory access time is roughly 800 ns (nanoseconds). On a 50 MHz 80486, the memory access time is slightly less than 20 ns. Note that the memory access time for the 80486 is 40 times faster than the 8088/8086. This is because the 80486's clock frequency is ten times faster and it uses one-fourth the clock cycles to access memory.

When reading from memory, the memory access time is the amount of time from the point that the CPU places an address on the address bus and the CPU takes the data off the data bus. On an 80486 CPU with a one cycle memory access time, a read looks something like shown in Figure 3.11. Writing data to memory is similar (see Figure 3.11).

Note that the CPU doesn't wait for memory. The access time is specified by the clock frequency. If the memory subsystem doesn't work fast enough, the CPU will read garbage data on a memory read operation and will not properly store the data on a memory write operation. This will surely cause the system to fail.

Memory devices have various ratings, but the two major ones are capacity and speed (access time). Typical dynamic RAM (random access memory) devices have capacities of four (or more) megabytes and speeds of 50-100 ns. You can buy bigger or faster devices, but they are much more expensive. A typical 33 MHz 80486 system uses 70 ns memory devices.

Wait just a second here! At 33 MHz the clock period is roughly 33 ns. How can a system designer get away with using 70 ns memory? The answer is *wait states.*

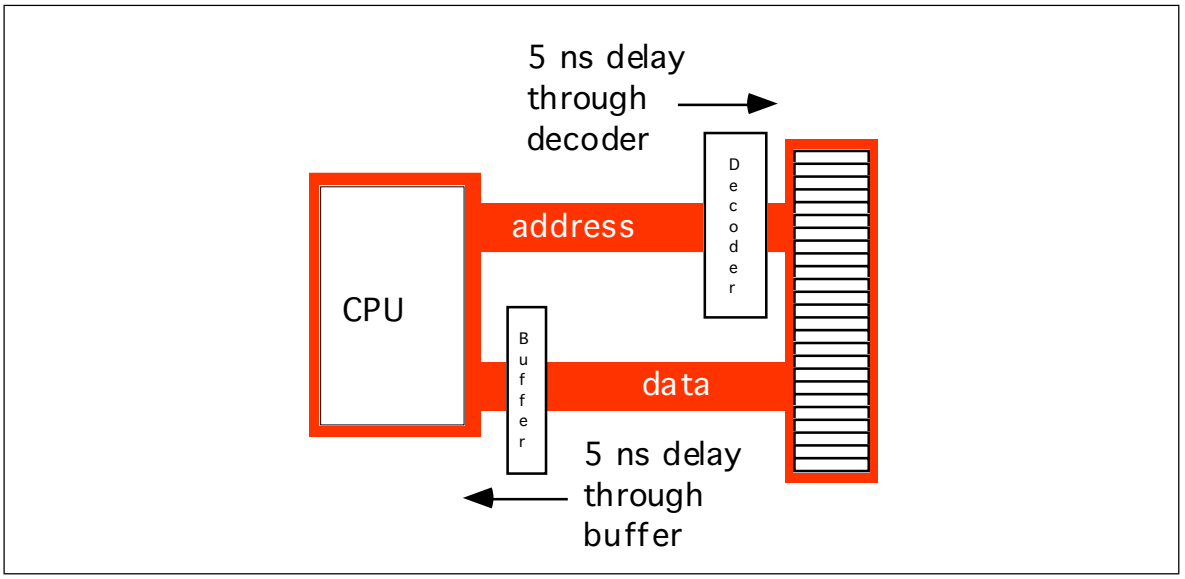


Figure 3.13 Decoding and Buffing Delays

3.2.3 Wait States

A wait state is nothing more than an extra clock cycle to give some device time to complete an operation. For example, a 50 MHz 80486 system has a 20 ns clock period. This implies that you need 20 ns memory. In fact, the situation is worse than this. In most computer systems there is additional circuitry between the CPU and memory: decoding and buffering logic. This additional circuitry introduces additional delays into the system (see Figure 3.13). In this diagram, the system loses 10ns to buffering and decoding. So if the CPU needs the data back in 20 ns, the memory must respond in less than 10 ns.

You can actually buy 10ns memory. However, it is very expensive, bulky, consumes a lot of power, and generates a lot of heat. These are bad attributes. Supercomputers use this type of memory. However, supercomputers also cost millions of dollars, take up entire rooms, require special cooling, and have giant power supplies. Not the kind of stuff you want sitting on your desk.

If cost-effective memory won't work with a fast processor, how do companies manage to sell fast PCs? One part of the answer is the wait state. For example, if you have a 20 MHz processor with a memory cycle time of 50 ns and you lose 10 ns to buffering and decoding, you'll need 40 ns memory. What if you can only afford 80 ns memory in a 20 MHz system? Adding a wait state to extend the memory cycle to 100 ns (two clock cycles) will solve this problem. Subtracting 10ns for the decoding and buffering leaves 90 ns. Therefore, 80 ns memory will respond well before the CPU requires the data.

Almost every general purpose CPU in existence provides a signal on the control bus to allow the insertion of wait states. Generally, the decoding circuitry asserts this line to delay one additional clock period, if necessary. This gives the memory sufficient access time, and the system works properly (see Figure 3.14).

Sometimes a single wait state is not sufficient. Consider the 80486 running at 50 MHz. The normal memory cycle time is less than 20 ns. Therefore, less than 10 ns are available after subtracting decoding and buffering time. If you are using 60 ns memory in the system, adding a single wait state will not do the trick. Each wait state gives you 20 ns, so with a single wait state you would need 30 ns memory. To work with 60 ns memory you would need to add *three* wait states (zero wait states = 10 ns, one wait state = 30 ns, two wait states = 50 ns, and three wait states = 70 ns).

Needless to say, from the system performance point of view, wait states are *not* a good thing. While the CPU is waiting for data from memory it cannot operate on that data.

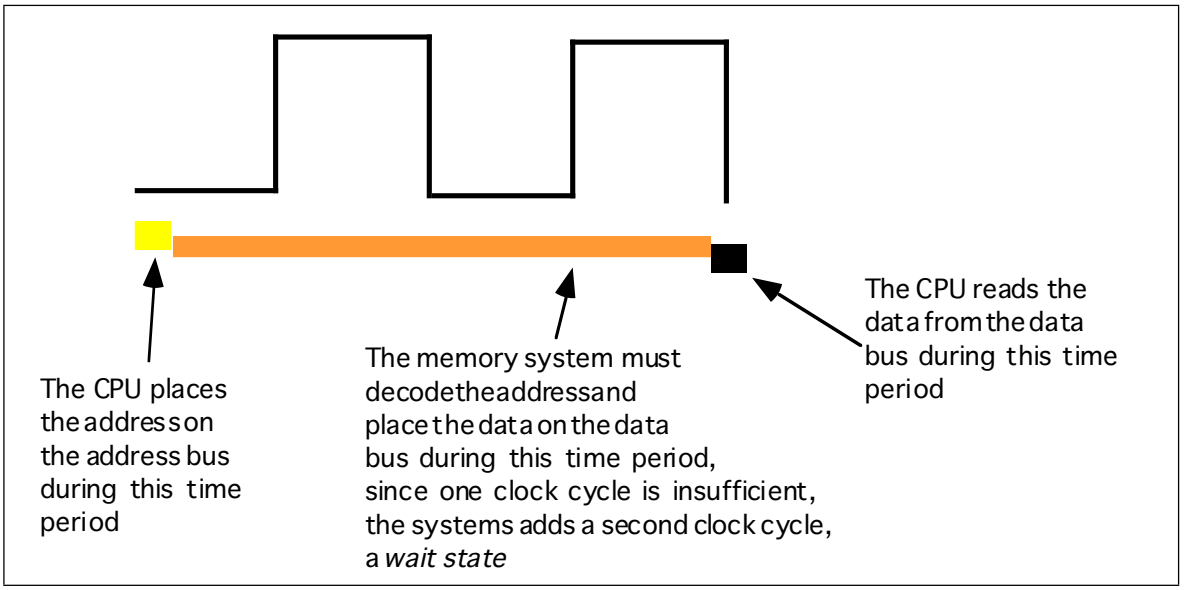


Figure 3.14 Inserting a Wait State into a Memory Read Operation

Adding a single wait state to a memory cycle on an 80486 CPU *doubles* the amount of time required to access the data. This, in turn, *halves* the speed of the memory access. Running with a wait state on every memory access is almost like cutting the processor clock frequency in half. You're going to get a lot less work done in the same amount of time.

You've probably seen the ads. "80386DX, 33 MHz, 8 megabytes 0 wait state RAM... only $1,000!" If you look closely at the specs you'll notice that the manufacturer is using 80 ns memory. How can they build systems which run at 33 MHz and have zero wait states? Easy. They lie.

There is no way an 80386 can run at 33 MHz, executing an arbitrary program, without ever inserting a wait state. It is flat out impossible. However, it is quite possible to design a memory subsystem which *under certain, special, circumstances* manages to operate without wait states part of the time. Most marketing types figure if their system *ever* operates at zero wait states, they can make that claim in their literature. Indeed, most marketing types have no idea what a wait state is other than it's bad and having zero wait states is something to brag about.

However, we're not doomed to slow execution because of added wait states. There are several tricks hardware designers can play to achieve zero wait states *most* of the time. The most common of these is the use of *cache* (pronounced "cash") memory.

3.2.4 Cache Memory

If you look at a typical program (as many researchers have), you'll discover that it tends to access the same memory locations repeatedly. Furthermore, you also discover that a program often accesses adjacent memory locations. The technical names given to this phenomenon are *temporal locality of reference* and *spatial locality of reference*. When exhibiting spatial locality, a program accesses neighboring memory locations. When displaying temporal locality of reference a program repeatedly accesses the same memory location during a short time period. Both forms of locality occur in the following Pascal code segment:

```
for i := 0 to 10 do
        A [i] := 0;
```

There are two occurrences each of spatial and temporal locality of reference within this loop. Let's consider the obvious ones first.

In the Pascal code above, the program references the variable i several times. The for loop compares i against 10 to see if the loop is complete. It also increments i by one at the bottom of the loop. The assignment statement also uses i as an array index. This shows temporal locality of reference in action since the CPU accesses i at three points in a short time period.

This program also exhibits spatial locality of reference. The loop itself zeros out the elements of array A by writing a zero to the first location in A, then to the second location in A, and so on. Assuming that Pascal stores the elements of A into consecutive memory locations[6], each loop iteration accesses adjacent memory locations.

There is an additional example of temporal and spatial locality of reference in the Pascal example above, although it is not so obvious. Computer *instructions* which tell the system to do the specified task also appear in memory. These instructions appear sequentially in memory – the spatial locality part. The computer also executes these instructions repeatedly, once for each loop iteration – the temporal locality part.

If you look at the execution profile of a typical program, you'd discover that the program typically executes less than half the statements. Generally, a typical program might only use 10-20% of the memory allotted to it. At any one given time, a one megabyte program might only access four to eight kilobytes of data and code. So if you paid an outrageous sum of money for expensive zero wait state RAM, you wouldn't be using most of it at any one given time! Wouldn't it be nice if you could buy a small amount of fast RAM and dynamically reassign its address(es) as the program executes?

This is exactly what cache memory does for you. Cache memory sits between the CPU and main memory. It is a small amount of very fast (zero wait state) memory. Unlike normal memory, the bytes appearing within a cache do not have fixed addresses. Instead, cache memory can reassign the address of a data object. This allows the system to keep recently accessed values in the cache. Addresses which the CPU has never accessed or hasn't accessed in some time remain in main (slow) memory. Since most memory accesses are to recently accessed variables (or to locations near a recently accessed location), the data generally appears in cache memory.

Cache memory is not perfect. Although a program may spend considerable time executing code in one place, eventually it will call a procedure or wander off to some section of code outside cache memory. In such an event the CPU has to go to main memory to fetch the data. Since main memory is slow, this will require the insertion of wait states.

A cache *hit* occurs whenever the CPU accesses memory and finds the data in the cache. In such a case the CPU can usually access data with zero wait states. A cache *miss* occurs if the CPU accesses memory and the data is not present in cache. Then the CPU has to read the data from main memory, incurring a performance loss. To take advantage of locality of reference, the CPU copies data into the cache whenever it accesses an address not present in the cache. Since it is likely the system will access that same location shortly, the system will save wait states by having that data in the cache.

As described above, cache memory handles the temporal aspects of memory access, but not the spatial aspects. Caching memory locations *when you access them* won't speed up the program if you constantly access consecutive locations (spatial locality of reference). To solve this problem, most caching systems read several consecutive bytes from memory when a cache miss occurs[7]. The 80486, for example, reads 16 bytes at a shot upon a cache miss. If you read 16 bytes, why read them in blocks rather than as you need them? As it turns out, most memory chips available today have special modes which let you quickly access several consecutive memory locations on the chip. The cache exploits this capability to reduce the average number of wait states needed to access memory.

If you write a program that randomly accesses memory, using a cache might actually slow you down. Reading 16 bytes on each cache miss is expensive if you only access a few

6. It does, see "Memory Layout and Access" on page 145.
7. Engineers call this block of data a cache *line*.

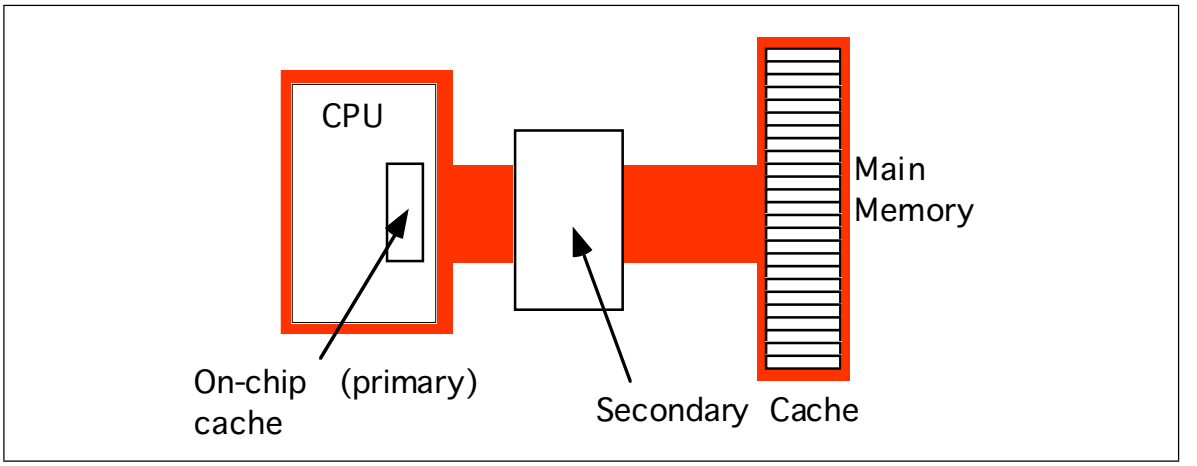


Figure 3.15 A Two Level Caching System

bytes in the corresponding cache line. Nonetheless, cache memory systems work quite well.

It should come as no surprise that the ratio of cache hits to misses increases with the size (in bytes) of the cache memory subsystem. The 80486 chip, for example, has 8,192 bytes of on-chip cache. Intel claims to get an 80-95% hit rate with this cache (meaning 80-95% of the time the CPU finds the data in the cache). This sounds very impressive. However, if you play around with the numbers a little bit, you'll discover it's not all *that* impressive. Suppose we pick the 80% figure. Then one out of every five memory accesses, on the average, will not be in the cache. If you have a 50 MHz processor and a 90 ns memory access time, four out of five memory accesses require only one clock cycle (since they are in the cache) and the fifth will require about 10 wait states[8]. Altogether, the system will require 15 clock cycles to access five memory locations, or three clock cycles per access, on the average. That's equivalent to two wait states added to every memory access. Now do you believe that your machine runs at zero wait states?

There are a couple of ways to improve the situation. First, you can add more cache memory. This improves the cache hit ratio, reducing the number of wait states. For example, increasing the hit ratio from 80% to 90% lets you access 10 memory locations in 20 cycles. This reduces the average number of wait states per memory access to one wait state – a substantial improvement. Alas, you can't pull an 80486 chip apart and solder more cache onto the chip. However, the 80586/Pentium CPU has a significantly larger cache than the 80486 and operates with fewer wait states.

Another way to improve performance is to build a *two-level* caching system. Many 80486 systems work in this fashion. The first level is the on-chip 8,192 byte cache. The next level, between the on-chip cache and main memory, is a secondary cache built on the computer system circuit board (see Figure 3.15).

A typical secondary cache contains anywhere from 32,768 bytes to one megabyte of memory. Common sizes on PC subsystems are 65,536 and 262,144 bytes of cache.

You might ask, "Why bother with a two-level cache? Why not use a 262,144 byte cache to begin with?" Well, the secondary cache generally does not operate at zero wait states. The circuitry to support 262,144 bytes of 10 ns memory (20 ns total access time) would be *very* expensive. So most system designers use slower memory which requires one or two wait states. This is still *much* faster than main memory. Combined with the on-chip cache, you can get better performance from the system.

8. Ten wait states were computed as follows: five clock cycles to read the first four bytes (10+20+20+20+20=90). However, the cache always reads 16 consecutive bytes. Most memory subsystems let you read consecutive addresses in about 40 ns after accessing the first location. Therefore, the 80486 will require an additional six clock cycles to read the remaining three double words. The total is 11 clock cycles or 10 wait states.

Consider the previous example with an 80% hit ratio. If the secondary cache requires two cycles for each memory access and three cycles for the first access, then a cache miss on the on-chip cache will require a total of six clock cycles. All told, the average system performance will be two clocks per memory access. Quite a bit faster than the three required by the system without the secondary cache. Furthermore, the secondary cache can update its values in parallel with the CPU. So the number of cache misses (which affect CPU performance) goes way down.

You're probably thinking, "So far this all sounds interesting, but what does it have to do with programming?" Quite a bit, actually. By writing your program carefully to take advantage of the way the cache memory system works, you can improve your program's performance. By colocating variables you commonly use together in the same cache line, you can force the cache system to load these variables as a group, saving extra wait states on each access.

If you organize your program so that it tends to execute the same sequence of instructions repeatedly, it will have a high degree of temporal locality of reference and will, therefore, execute faster.

3.3 The 886, 8286, 8486, and 8686 "Hypothetical" Processors

To understand how to improve system performance, it's time to explore the internal operation of the CPU. Unfortunately, the processors in the 80x86 family are complex beasts. Discussing their internal operation would probably cause more confusion than enlightenment. So we will use the 886, 8286, 8486, and 8686 processors (the "x86" processors). These "paper processors" are extreme simplifications of various members of the 80x86 family. They highlight the important architectural features of the 80x86.

The 886, 8286, 8486, and 8686 processors are all identical except for the way they execute instructions. They all have the same *register set*, and they "execute" the same *instruction set*. That sentence contains some new ideas; let's attack them one at a time.

3.3.1 CPU Registers

CPU registers are *very* special memory locations constructed from flip-flops. They are not part of main memory; the CPU implements them on-chip. Various members of the 80x86 family have different register sizes. The 886, 8286, 8486, and 8686 (x86 from now on) CPUs have exactly four registers, all 16 bits wide. All arithmetic and location operations occur in the CPU registers.

Because the x86 processor has so few registers, we'll give each register its own name and refer to it by that name rather than its address. The names for the x86 registers are

```
AX      –The accumulator register
BX      –The base address register
CX      –The count register
DX      –The data register
```

Besides the above registers, which are visible to the programmer, the x86 processors also have an *instruction pointer* register which contains the address of the next instruction to execute. There is also a *flags* register that holds the result of a comparison. The flags register remembers if one value was less than, equal to, or greater than another value.

Because registers are on-chip and handled specially by the CPU, they are much faster than memory. Accessing a memory location requires one or more clock cycles. Accessing data in a register usually takes zero clock cycles. Therefore, you should try to keep variables in the registers. Register sets are very small and most registers have special purposes which limit their use as variables, but they are still an excellent place to store temporary data.

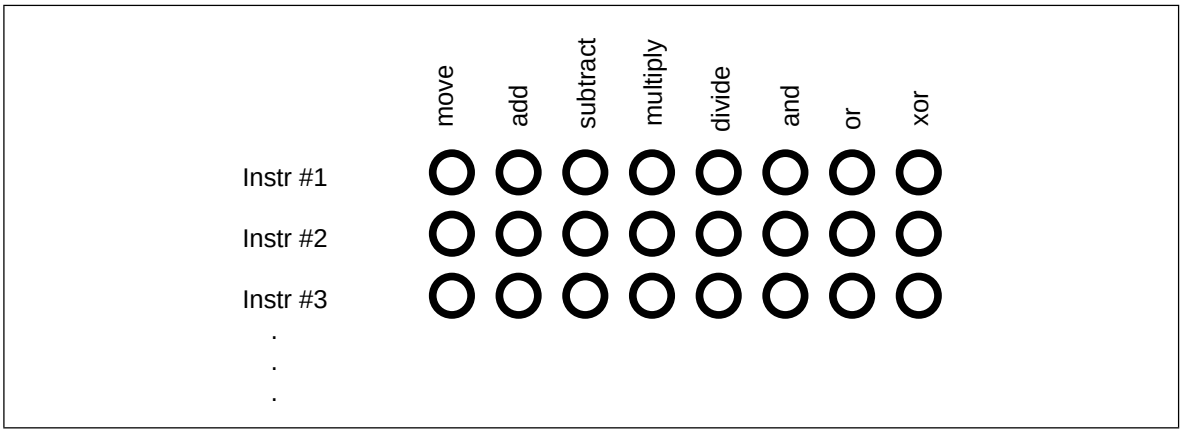


Figure 3.16 Patch Panel Programming

3.3.2 The Arithmetic & Logical Unit

The arithmetic and logical unit (ALU) is where most of the action takes place inside the CPU. For example, if you want to add the value five to the AX register, the CPU:

- Copies the value from AX into the ALU,
- Sends the value five to the ALU,
- Instructs the ALU to add these two values together,
- Moves the result back into the AX register.

3.3.3 The Bus Interface Unit

The bus interface unit (BIU) is responsible for controlling the address and data busses when accessing main memory. If a cache is present on the CPU chip then the BIU is also responsible for accessing data in the cache.

3.3.4 The Control Unit and Instruction Sets

A fair question to ask at this point is "How exactly does a CPU perform assigned chores?" This is accomplished by giving the CPU a fixed set of commands, or *instructions*, to work on. Keep in mind that CPU designers construct these processors using logic gates to execute these instructions. To keep the number of logic gates to a reasonably small set (tens or hundreds of thousands), CPU designers must necessarily restrict the number and complexity of the commands the CPU recognizes. This small set of commands is the CPU's *instruction set*.

Programs in early (pre-Von Neumann) computer systems were often "hard-wired" into the circuitry. That is, the computer's wiring determined what problem the computer would solve. One had to rewire the circuitry in order to change the program. A very difficult task. The next advance in computer design was the *programmable* computer system, one that allowed a computer programmer to easily "rewire" the computer system using a sequence of sockets and plug wires. A computer program consisted of a set of rows of holes (sockets), each row representing one operation during the execution of the program. The programmer could select one of several instructions by plugging a wire into the particular socket for the desired instruction (see Figure 3.16). Of course, a major difficulty with this scheme is that the number of possible instructions is severely limited by the number of sockets one could physically place on each row. However, CPU designers quickly discovered that with a small amount of additional logic circuitry, they could reduce the number of sockets required from n holes for n instructions to $\log_2(n)$ holes for n instructions. They did this by assigning a *numeric* code to each instruction and then

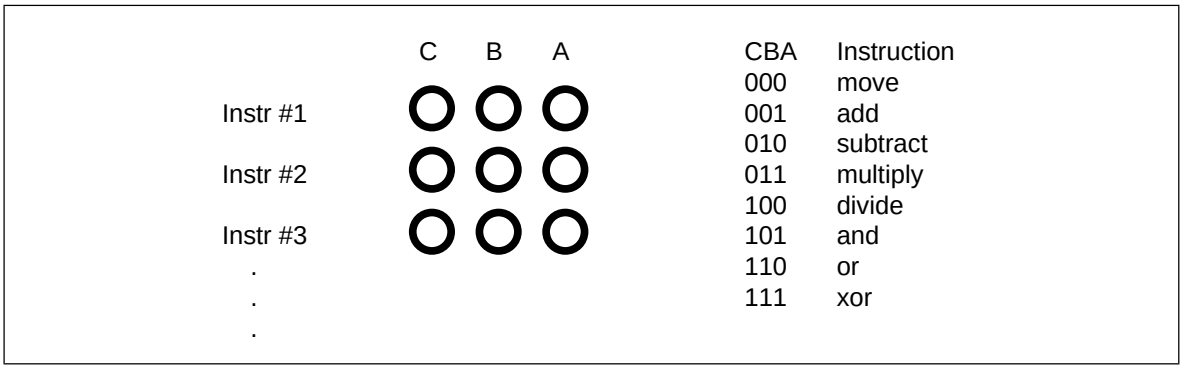


Figure 3.17 Encoding Instructions

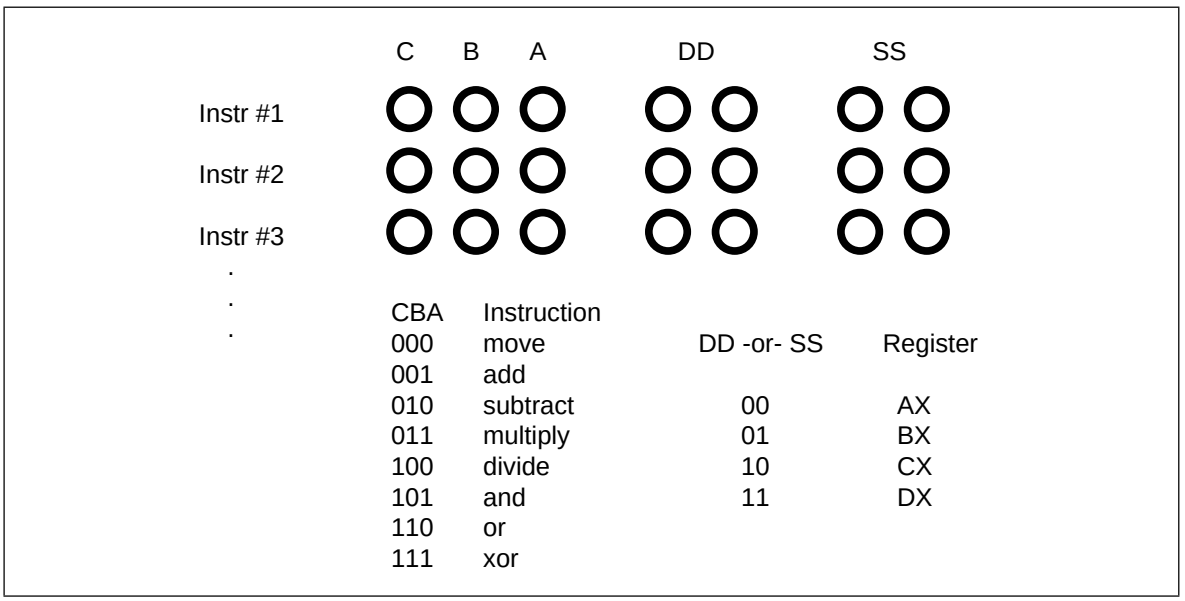


Figure 3.18 Encoding Instructions with Source and Destination Fields

encode that instruction as a binary number using $\log_2(n)$ holes (see Figure 3.17). This addition requires eight logic functions to decode the A, B, and C bits from the patch panel, but the extra circuitry is well worth the cost because it reduces the number of sockets that must be repeated for each instruction.

Of course, many CPU instructions are not stand-alone. For example, the move instruction is a command that moves data from one location in the computer to another (e.g., from one register to another). Therefore, the move instruction requires two operands: a *source operand* and a *destination operand.* The CPU's designer usually encodes these source and destination operands as part of the machine instruction, certain sockets correspond to the source operand and certain sockets correspond to the destination operand. Figure 3.17 shows one possible combination of sockets to handle this. The move instruction would move data from the source register to the destination register, the add instruction would add the value of the source register to the destination register, etc.

One of the primary advances in computer design that the VNA provides is the concept of a *stored program.* One big problem with the patch panel programming method is that the number of program steps (machine instructions) is limited by the number of rows of sockets available on the machine. John Von Neumann and others recognized a relationship between the sockets on the patch panel and bits in memory; they figured they could store the binary equivalents of a machine program in main memory and fetch each program from memory, load it into a special *decoding register* that connected directly to the instruction decoding circuitry of the CPU.

The trick, of course, was to add yet more circuitry to the CPU. This circuitry, the *control unit* (CU), fetches instruction codes (also known as *operation codes* or *opcodes*) from memory and moves them to the instruction decoding register. The control unit contains a special registers, the *instruction pointer* that contains the address of an executable instruction. The control unit fetches this instruction's code from memory and places it in the decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution, and so on.

When designing an instruction set, the CPU's designers generally choose opcodes that are a multiple of eight bits long so the CPU can easily fetch complete instructions from memory. The goal of the CPU's designer is to assign an appropriate number of bits to the instruction class field (move, add, subtract, etc.) and to the operand fields. Choosing more bits for the instruction field lets you have more instructions, choosing additional bits for the operand fields lets you select a larger number of operands (e.g., memory locations or registers). There are additional complications. Some instructions have only one operand or, perhaps, they don't have any operands at all. Rather than waste the bits associated with these fields, the CPU designers often reuse these fields to encode additional opcodes, once again with some additional circuitry. The Intel 80x86 CPU family takes this to an extreme with instructions ranging from one to about ten bytes long. Since this is a little too difficult to deal with at this early stage, the x86 CPUs will use a different, much simpler, encoding scheme.

3.3.5 The x86 Instruction Set

The x86 CPUs provide 20 basic instruction classes. Seven of these instructions have two operands, eight of these instructions have a single operand, and five instructions have no operands at all. The instructions are mov (two forms), add, sub, cmp, and, or, not, je, jne, jb, jbe, ja, jae, jmp, brk, iret, halt, get, and put. The following paragraphs describe how each of these work.

The mov instruction is actually two instruction classes merged into the same instruction. The two forms of the mov instruction take the following forms:

```
mov     reg, reg/memory/constant
mov     memory, reg
```

where reg is any of ax, bx, cx, or dx; constant is a numeric constant (using hexadecimal notation), and memory is an operand specifying a memory location. The next section describes the possible forms the memory operand can take. The "reg/memory/constant" operand tells you that this particular operand may be a register, memory location, or a constant.

The *arithmetic and logical instructions* take the following forms:

```
add     reg, reg/memory/constant
sub     reg, reg/memory/constant
cmp     reg, reg/memory/constant
and     reg, reg/memory/constant
or      reg, reg/memory/constant
not     reg/memory
```

The add instruction adds the value of the second operand to the first (register) operand, leaving the sum in the first operand. The sub instruction subtracts the value of the second operand from the first, leaving the difference in the first operand. The cmp instruction compares the first operand against the second and saves the result of this comparison for use with one of the conditional jump instructions (described in a moment). The and and or instructions compute the corresponding bitwise logical operation on the two operands and store the result into the first operand. The not instruction inverts the bits in the single memory or register operand.

The *control transfer instructions* interrupt the sequential execution of instructions in memory and transfer control to some other point in memory either unconditionally, or

after testing the result of the previous cmp instruction. These instructions include the following:

```
ja      dest        -- Jump if above
jae     dest        -- Jump if above or equal
jb      dest        -- Jump if below
jbe     dest        -- Jump if below or equal
je      dest        -- Jump if equal
jne     dest        -- Jump if not equal
jmp     dest        -- Unconditional jump
iret                -- Return from an interrupt
```

The first six instructions in this class let you check the result of the previous cmp instruction for greater than, greater or equal, less than, less or equal, equality, or inequality[9]. For example, if you compare the ax and bx registers with the cmp instruction and execute the ja instruction, the x86 CPU will jump to the specified destination location if ax was greater than bx. If ax is not greater than bx, control will fall through to the next instruction in the program. The jmp instruction unconditionally transfers control to the instruction at the destination address. The iret instruction returns control from an *interrupt service routine*, which we will discuss later.

The get and put instructions let you read and write integer values. Get will stop and prompt the user for a hexadecimal value and then store that value into the ax register. Put displays (in hexadecimal) the value of the ax register.

The remaining instructions do not require any operands, they are halt and brk. Halt terminates program execution and brk stops the program in a state that it can be restarted.

The x86 processors require a unique opcode for every different instruction, not just the instruction classes. Although "mov ax, bx" and "mov ax, cx" are both in the same class, they must have different opcodes if the CPU is to differentiate them. However, before looking at all the possible opcodes, perhaps it would be a good idea to learn about all the possible operands for these instructions.

3.3.6 Addressing Modes on the x86

The x86 instructions use five different operand types: registers, constants, and three memory addressing schemes. Each form is called an *addressing mode*. The x86 processors support the *register* addressing mode[10], the *immediate* addressing mode, the *indirect* addressing mode, the *indexed* addressing mode, and the *direct* addressing mode. The following paragraphs explain each of these modes.

Register operands are the easiest to understand. Consider the following forms of the mov instruction:

```
mov     ax, ax
mov     ax, bx
mov     ax, cx
mov     ax, dx
```

The first instruction accomplishes absolutely nothing. It copies the value from the ax register back into the ax register. The remaining three instructions copy the value of bx, cx and dx into ax. Note that the original values of bx, cx, and dx remain the same. The first operand (the *destination*) is not limited to ax; you can move values to any of these registers.

Constants are also pretty easy to deal with. Consider the following instructions:

```
mov     ax, 25
mov     bx, 195
mov     cx, 2056
mov     dx, 1000
```

9. The x86 processors only performed *unsigned* comparisons.

10. Technically, registers do not have an address, but we apply the term *addressing mode* to registers nonetheless.

These instructions are all pretty straightforward; they load their respective registers with the specified hexadecimal constant[11].

There are three addressing modes which deal with accessing data in memory. These addressing modes take the following forms:

```
mov     ax, [1000]
mov     ax, [bx]
mov     ax, [1000+bx]
```

The first instruction above uses the *direct* addressing mode to load ax with the 16 bit value stored in memory starting at location 1000 hex.

The `mov ax, [bx]` instruction loads ax from the memory location specified by the contents of the bx register. This is an *indirect* addressing mode. Rather than using the value in bx, this instruction accesses to the memory location whose address appears in bx. Note that the following two instructions:

```
mov     bx, 1000
mov     ax, [bx]
```

are equivalent to the single instruction:

```
mov     ax, [1000]
```

Of course, the second sequence is preferable. However, there are many cases where the use of indirection is faster, shorter, and better. We'll see some examples of this when we look at the individual processors in the x86 family a little later.

The last memory addressing mode is the *indexed* addressing mode. An example of this memory addressing mode is

```
mov     ax, [1000+bx]
```

This instruction adds the contents of bx with 1000 to produce the address of the memory value to fetch. This instruction is useful for accessing elements of arrays, records, and other data structures.

3.3.7 Encoding x86 Instructions

Although we could arbitrarily assign opcodes to each of the x86 instructions, keep in mind that a real CPU uses logic circuitry to decode the opcodes and act appropriately on them. A typical CPU opcode uses a certain number of bits in the opcode to denote the instruction class (e.g., mov, add, sub), and a certain number of bits to encode each of the operands. Some systems (e.g., CISC, or Complex Instruction Set Computers) encode these fields in a very complex fashion producing very compact instructions. Other systems (e.g., RISC, or Reduced Instruction Set Computers) encode the opcodes in a very simple fashion even if it means wasting some bits in the opcode or limiting the number of operations. The Intel 80x86 family is definitely CISC and has one of the most complex opcode decoding schemes ever devised. The whole purpose for the hypothetical x86 processors is to present the concept of instruction encoding without the attendant complexity of the 80x86 family, while still demonstrating CISC encoding.

A typical x86 instruction takes the form shown in Figure 3.19. The basic instruction is either one or three bytes long. The instruction opcode consists of a single byte that contains three fields. The first field, the H.O. three bits, defines the instruction class. This provides eight combinations. As you may recall, there are 20 instruction classes; we cannot encode 20 instruction classes with three bits, so we'll have to pull some tricks to handle the other classes. As you can see in Figure 3.19, the basic opcode encodes the mov instructions (two classes, one where the rr field specifies the destination, one where the mmm field specifies the destination), the add, sub, cmp, and, and or instructions. There is one

11. All numeric constants on the x86 are given in hexadecimal. The "h" suffix is not necessary.

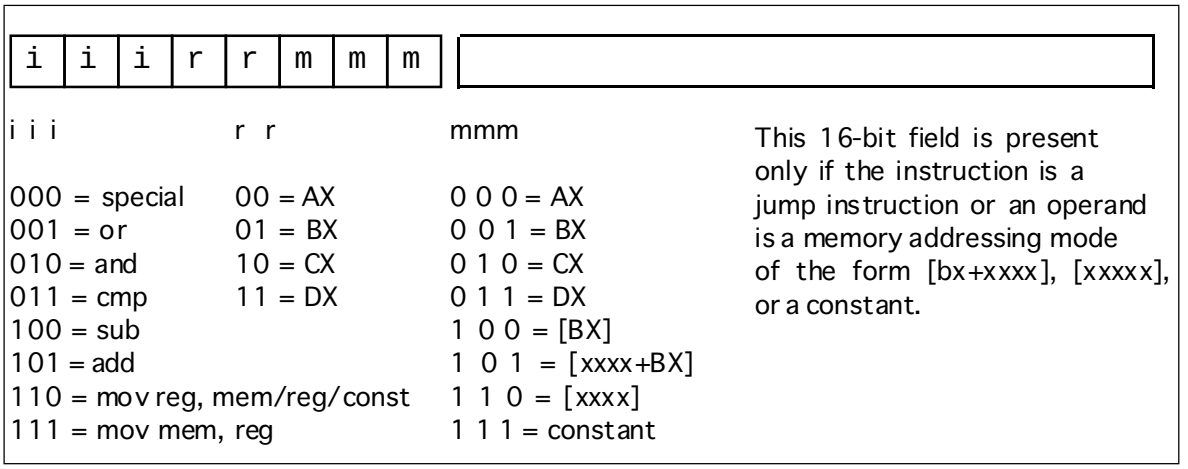


Figure 3.19 Basic x86 Instruction Encoding.

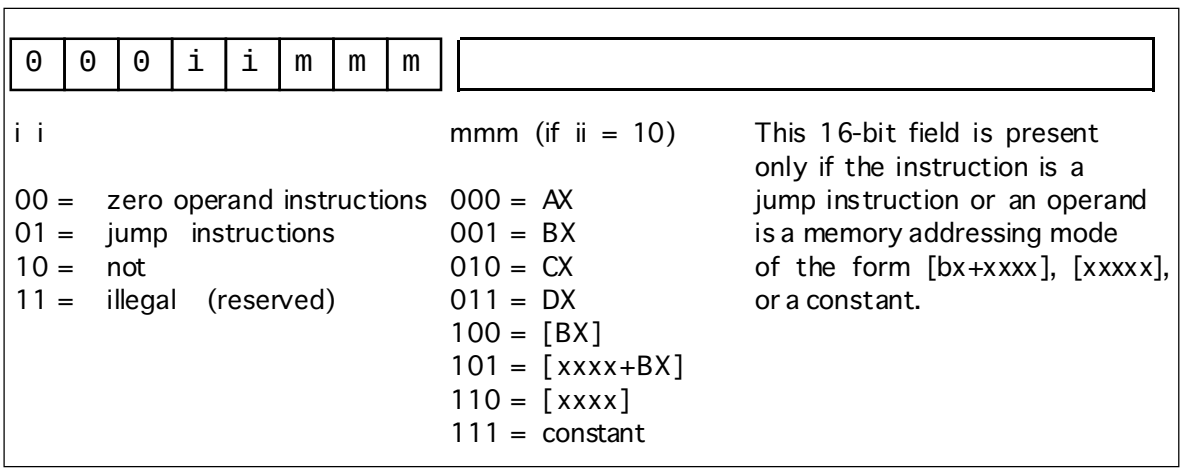


Figure 3.20 Single Operand Instruction Encodings

additional class: special. The special instruction class provides a mechanism that allows us to expand the number of available instruction classes, we will return to this class shortly.

To determine a particular instruction's opcode, you need only select the appropriate bits for the iii, rr, and mmm fields. For example, to encode the mov ax, bx instruction you would select iii=110 (mov reg, reg), rr=00 (ax), and mmm=001 (bx). This produces the one-byte instruction 11000001 or 0C0h.

Some x86 instructions require more than one byte. For example, the instruction mov ax, [1000] loads the ax register from memory location 1000. The encoding for the opcode is 11000110 or 0C6h. However, the encoding for mov ax,[2000]'s opcode is also 0C6h. Clearly these two instructions do different things, one loads the ax register from memory location 1000h while the other loads the ax register from memory location 2000. To encode an address for the [xxxx] or [xxxx+bx] addressing modes, or to encode the constant for the immediate addressing mode, you must follow the opcode with the 16-bit address or constant, with the L.O. byte immediately following the opcode in memory and the H.O. byte after that. So the three byte encoding for mov ax, [1000] would be 0C6h, 00h, 10h[12] and the three byte encoding for mov ax, [2000] would be 0C6h, 00h, 20h.

The special opcode allows the x86 CPU to expand the set of available instructions. This opcode handles several zero and one-operand instructions as shown in Figure 3.20 and Figure 3.21.

12. Remember, all numeric constants are hexadecimal.

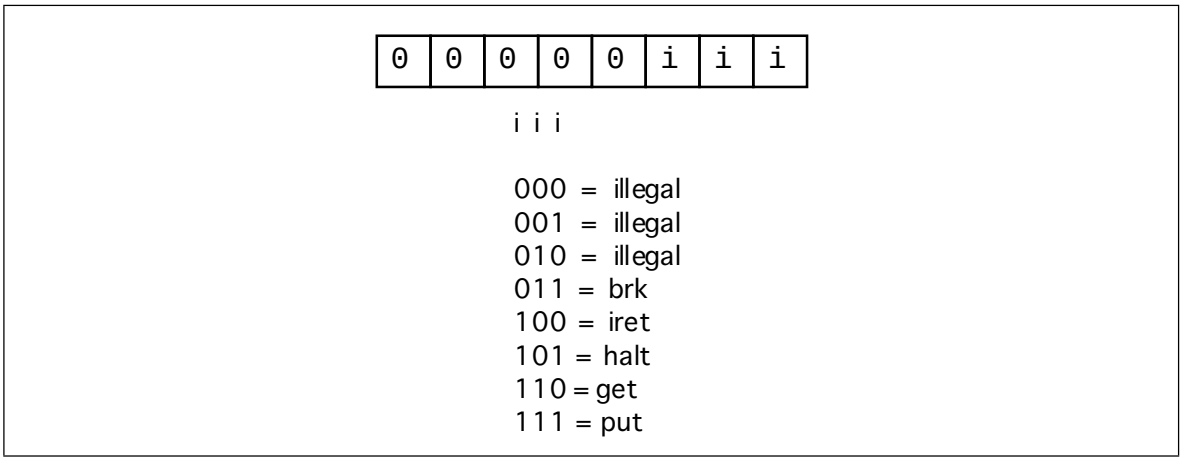


Figure 3.21 Zero Operand Instruction Encodings

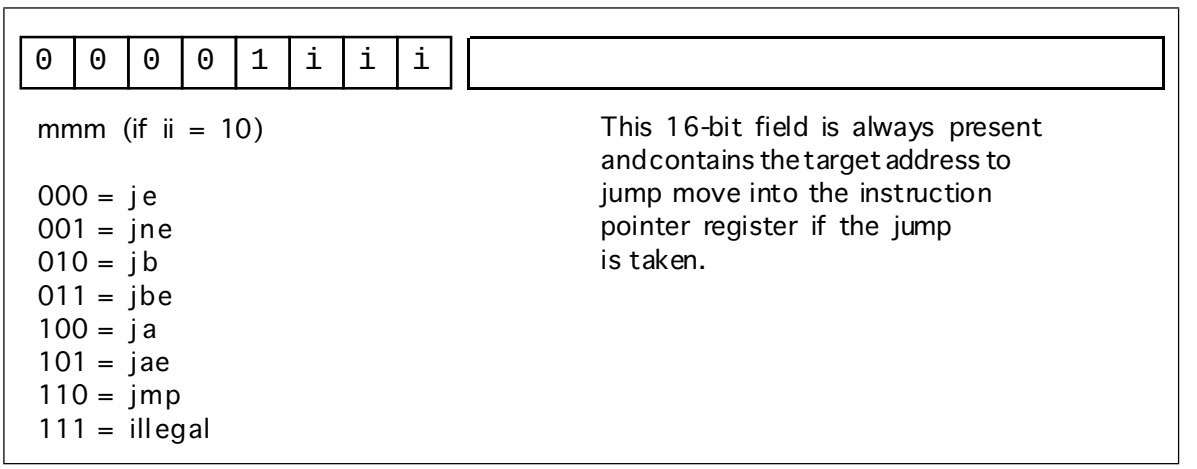


Figure 3.22 Jump Instruction Encodings

There are four one-operand instruction classes. The first encoding (00) further expands the instruction set with a set of zero-operand instructions (see Figure 3.21). The second opcode is also an expansion opcode that provides all the x86 *jump* instructions (see Figure 3.22). The third opcode is the not instruction. This is the bitwise logical not operation that inverts all the bits in the destination register or memory operand. The fourth single-operand opcode is currently unassigned. Any attempt to execute this opcode will halt the processor with an illegal instruction error. CPU designers often reserve unassigned opcodes like this one to extend the instruction set at a future date (as Intel did when moving from the 80286 processor to the 80386).

There are seven jump instructions in the x86 instruction set. They all take the following form:

```
jxx     address
```

The jmp instruction copies the 16-bit immediate value (address) following the opcode into the IP register. Therefore, the CPU will fetch the next instruction from this target address; effectively, the program "jumps" from the point of the jmp instruction to the instruction at the target address.

The jmp instruction is an example of an *unconditional jump instruction*. It always transfers control to the target address. The remaining six instructions are *conditional jump instructions*. They test some condition and jump if the condition is true; they fall through to the next instruction if the condition is false. These six instructions, ja, jae, jb, jbe, je, and jne let you test for greater than, greater than or equal, less than, less than or equal, equality, and inequality. You would normally execute these instructions immediately after a cmp

instruction since it sets the less than and equality flags that the conditional jump instructions test. Note that there are eight possible jump opcodes, but the x86 uses only seven of them. The eighth opcode is another illegal opcode.

The last group of instructions, the zero operand instructions, appear in Figure 3.21. Three of these instructions are illegal instruction opcodes. The brk (break) instruction pauses the CPU until the user manually restarts it. This is useful for pausing a program during execution to observe results. The iret (interrupt return) instruction returns control from an *interrupt service routine*. We will discuss interrupt service routines later. The halt program terminates program execution. The get instruction reads a hexadecimal value from the user and returns this value in the ax register; the put instruction outputs the value in the ax register.

3.3.8 Step-by-Step Instruction Execution

The x86 CPUs do *not* complete execution of an instruction in a single clock cycle. The CPU executes several steps for each instruction. For example, the CU issues the following commands to execute the mov reg, reg/memory/constant instruction:

- Fetch the instruction byte from memory.
- Update the ip register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a 16-bit instruction operand from memory.
- If required, update ip to point beyond the operand.
- Compute the address of the operand, if required (i.e., bx+xxxx) .
- Fetch the operand.
- Store the fetched value into the destination register

A step-by-step description may help clarify what the CPU is doing. In the first step, the CPU fetches the instruction byte from memory. To do this, it copies the value of the ip register to the address bus and reads the byte at that address. This will take one clock cycle[13].

After fetching the instruction byte, the CPU updates ip so that it points at the next byte in the instruction stream. If the current instruction is a multibyte instruction, ip will now point at the operand for the instruction. If the current instruction is a single byte instruction, ip would be left pointing at the next instruction. This takes one clock cycle.

The next step is to decode the instruction to see what it does. This will tell the CPU, among other things, if it needs to fetch additional operand bytes from memory. This takes one clock cycle.

During decoding, the CPU determines the types of operands the instruction requires. If the instruction requires a 16 bit constant operand (i.e., if the *mmm* field is 101, 110, or 111) then the CPU fetches that constant from memory. This step may require zero, one, or two clock cycles. It requires zero cycles if there is no 16 bit operand; it requires one clock cycle if the 16 bit operand is word-aligned (that is, begins at an even address); it requires two clock cycles if the operand is not word aligned (that is, begins at an odd address).

If the CPU fetches a 16 bit memory operand, it must increment ip by two so that it points at the next byte following the operand. This operation takes zero or one clock cycles. Zero clock cycles if there is no operand; one if an operand is present.

Next, the CPU computes the address of the memory operand. This step is required only when the *mmm* field of the instruction byte is 101 or 100. If the *mmm* field contains 101, then the CPU computes the sum of the bx register and the 16 bit constant; this requires two cycles, one cycle to fetch bx's value, the other to computer the sum of bx and xxxx. If the *mmm* field contains 100, then the CPU fetches the value in bx for the memory

13. We will assume that clock cycles and memory cycles are equivalent.

address, this requires one cycle. If the *mmm* field does not contain 100 or 101, then this step takes zero cycles.

Fetching the operand takes zero, one, two, or three cycles depending upon the operand itself. If the operand is a constant (*mmm*=111), then this step requires zero cycles because we've already fetched this constant from memory in a previous step. If the operand is a register (*mmm* = 000, 001, 010, or 011) then this step takes one clock cycle. If this is a word aligned memory operand (*mmm*=100, 101, or 110) then this step takes two clock cycles. If it is an unaligned memory operand, it takes three clock cycles to fetch its value.

The last step to the mov instruction is to store the value into the destination location. Since the destination of the load instruction is always a register, this operation takes a single cycle.

Altogether, the mov instruction takes between five and eleven cycles, depending on its operands and their alignment (starting address) in memory.

The CPU does the following for the mov memory, reg instruction:

- Fetch the instruction byte from memory (one clock cycle).
- Update ip to point at the next byte (one clock cycle).
- Decode the instruction to see what it does (one clock cycle).
- If required, fetch an operand from memory (zero cycles if [bx] addressing mode, one cycle if [xxxx], [xxxx+bx], or xxxx addressing mode and the value xxxx immediately following the opcode starts on an even address, or two clock cycles if the value xxxx starts at an odd address).
- If required, update ip to point beyond the operand (zero cycles if no such operand, one clock cycle if the operand is present).
- Compute the address of the operand (zero cycles if the addressing mode is not [bx] or [xxxx+bx], one cycle if the addressing mode is [bx], or two cycles if the addressing mode is [xxxx+bx]).
- Get the value of the register to store (one clock cycle).
- Store the fetched value into the destination location (one cycle if a register, two cycles if a word-aligned memory operand, or three clock cycles if an odd-address aligned memory operand).

The timing for the last two items is different from the other mov because that instruction can read data from memory; this version of mov instruction "loads" its data from a register. This instruction takes five to eleven clock cycles to execute.

The add, sub, cmp, and, and or instructions do the following:

- Fetch the instruction byte from memory (one clock cycle).
- Update ip to point at the next byte (one clock cycle).
- Decode the instruction (one clock cycle).
- If required, fetch a constant operand from memory (zero cycles if [bx] addressing mode, one cycle if [xxxx], [xxxx+bx], or xxxx addressing mode and the value xxxx immediately following the opcode starts on an even address, or two clock cycles if the value xxxx starts at an odd address).
- If required, update ip to point beyond the constant operand (zero or one clock cycles).
- Compute the address of the operand (zero cycles if the addressing mode is not [bx] or [xxxx+bx], one cycle if the addressing mode is [bx], or two cycles if the addressing mode is [xxxx+bx]).
- Get the value of the operand and send it to the ALU (zero cycles if a constant, one cycle if a register, two cycles if a word-aligned memory operand, or three clock cycles if an odd-address aligned memory operand).
- Fetch the value of the first operand (a register) and send it to the ALU (one clock cycle).
- Instruct the ALU to add, subtract, compare, logically and, or logically or the values (one clock cycle).
- Store the result back into the first register operand (one clock cycle).

These instructions require between eight and seventeen clock cycles to execute.

The not instruction is similar to the above, but may be a little faster since it only has a single operand:

- Fetch the instruction byte from memory (one clock cycle).
- Update ip to point at the next byte (one clock cycle).
- Decode the instruction (one clock cycle).
- If required, fetch a constant operand from memory (zero cycles if [bx] addressing mode, one cycle if [xxxx] or [xxxx+bx] addressing mode and the value xxxx immediately following the opcode starts on an even address, or two clock cycles if the value xxxx starts at an odd address).
- If required, update ip to point beyond the constant operand (zero or one clock cycles).
- Compute the address of the operand (zero cycles if the addressing mode is not [bx] or [xxxx+bx], one cycle if the addressing mode is [bx], or two cycles if the addressing mode is [xxxx+bx]).
- Get the value of the operand and send it to the ALU (one cycle if a register, two cycles if a word-aligned memory operand, or three clock cycles if an odd-address aligned memory operand).
- Instruct the ALU to logically not the values (one clock cycle).
- Store the result back into the operand (one clock cycle if a register, two clock cycles if an even-aligned memory location, three cycles if odd-aligned memory location).

The not instruction takes six to fifteen cycles to execute.

The conditional jump instructions work as follows:

- Fetch the instruction byte from memory (one clock cycle).
- Update ip to point at the next byte (one clock cycle).
- Decode the instructions (one clock cycle).
- Fetch the target address operand from memory (one cycle if xxxx is at an even address, two clock cycles if at an odd address).
- Update ip to point beyond the address (one clock cycle).
- Test the “less than” and “equality” CPU flags (one cycle).
- If the flag values are appropriate for the particular conditional jump, the CPU copies the 16 bit constant into the ip register (zero cycles if no branch, one clock cycle if branch occurs).

The unconditional jump instruction is identical in operation to the mov reg, xxxx instruction except the destination register is the x86’s ip register rather than ax, bx, cx, or dx.

The brk, iret, halt, put, and get instructions are of no interest to us here. They appear in the instruction set mainly for programs and experiments. We can’t very well give them “cycle” counts since they may take an indefinite amount of time to complete their task.

3.3.9 The Differences Between the x86 Processors

All the x86 processors share the same instruction set, the same addressing modes, and execute their instructions using the same sequence of steps. So what’s the difference? Why not invent one processor rather than four?

The main reason for going through this exercise is to explain performance differences related to four hardware features: *pre-fetch queues*, *caches*, *pipelines* and *superscalar designs*. The 886 processor is an inexpensive “device” which doesn’t implement any of these fancy features. The 8286 processor implements the prefetch queue. The 8486 has a pre-fetch queue, a cache, and a pipeline. The 8686 has all of the above features with superscalar operation. By studying each of these processors you can see the benefits of each feature.

3.3.10 The 886 Processor

The 886 processor is the slowest member of the x86 family. Timings for each instruction were discussed in the previous sections. The mov instruction, for example, takes between five and twelve clock cycles to execute depending upon the operands. The following table provides the timing for the various forms of the instructions on the 886 processors.

Table 19: Execution Times for 886 Instructions

Instruction ⇒ Addressing Mode ⇓	mov (both forms)	add, sub, cmp, and, or,	not	jmp	jxx
reg, reg	5	7			
reg, xxxx	6-7	8-9			
reg, [bx]	7-8	9-10			
reg, [xxxx]	8-10	10-12			
reg, [xxxx+bx]	10-12	12-14			
[bx], reg	7-8				
[xxxx], reg	8-10				
[xxxx+bx], reg	10-12				
reg			6		
[bx]			9-11		
[xxxx]			10-13		
[xxxx+bx]			12-15		
xxxx				6-7	6-8

There are three important things to note from this. First, longer instructions take more time to execute. Second, instructions that do not reference memory generally execute faster; this is especially true if there are wait states associated with memory access (the table above assumes zero wait states). Finally, instructions using complex addressing modes run slower. Instructions which use register operands are shorter, do not access memory, and do not use complex addressing modes. *This is why you should attempt to keep your variables in registers.*

3.3.11 The 8286 Processor

The key to improving the speed of a processor is to perform operations in parallel. If, in the timings given for the 886, we were able to do two operations on each clock cycle, the CPU would execute instructions twice as fast when running at the same clock speed. However, simply deciding to execute two operations per clock cycle is not so easy. Many steps in the execution of an instruction share *functional units* in the CPU (functional units are groups of logic that perform a common operation, e.g., the ALU and the CU). A functional unit is only capable of one operation at a time. Therefore, you cannot do two operations that use the same functional unit concurrently (e.g., incrementing the ip register and adding two values together). Another difficulty with doing certain operations concurrently is that one operation may depend on the other's result. For example, the last two steps of the add instruction involve adding to values and then storing their sum. You cannot store the sum into a register until after you've computed the sum. There are also some other resources the CPU cannot share between steps in an instruction. For example, there

is only one data bus; the CPU cannot fetch an instruction opcode at the same time it is trying to store some data to memory. The trick in designing a CPU that executes several steps in parallel is to arrange those steps to reduce conflicts or add additional logic so the two (or more) operations can occur simultaneously by executing in different functional units.

Consider again the steps the mov reg, mem/reg/const instruction requires:

- Fetch the instruction byte from memory.
- Update the ip register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a 16-bit instruction operand from memory.
- If required, update ip to point beyond the operand.
- Compute the address of the operand, if required (i.e., bx+xxxx) .
- Fetch the operand.
- Store the fetched value into the destination register

The first operation uses the value of the ip register (so we cannot overlap incrementing ip with it) and it uses the bus to fetch the instruction opcode from memory. Every step that follows this one depends upon the opcode it fetches from memory, so it is unlikely we will be able to overlap the execution of this step with any other.

The second and third operations do not share any functional units, nor does decoding an opcode depend upon the value of the ip register. Therefore, we can easily modify the control unit so that it increments the ip register at the same time it decodes the instruction. This will shave one cycle off the execution of the mov instruction.

The third and fourth operations above (decoding and optionally fetching the 16-bit operand) do not look like they can be done in parallel since you must decode the instruction to determine if it the CPU needs to fetch a 16-bit operand from memory. However, we could design the CPU to go ahead and fetch the operand anyway, so that it's available if we need it. There is one problem with this idea, though, we must have the address of the operand to fetch (the value in the ip register) and if we must wait until we are done incrementing the ip register before fetching this operand. If we are incrementing ip at the same time we're decoding the instruction, we will have to wait until the next cycle to fetch this operand.

Since the next three steps are optional, there are several possible instruction sequences at this point:

#1 (step 4, step 5, step 6, and step 7) – e.g., mov ax, [1000+bx]

#2 (step 4, step 5, and step 7) – e.g., mov ax, [1000]

#3 (step 6 and step 7) – e.g., mov ax, [bx]

#4 (step 7) – e.g., mov ax, bx

In the sequences above, step seven always relies on the previous set in the sequence. Therefore, step seven cannot execute in parallel with any of the other steps. Step six also relies upon step four. Step five cannot execute in parallel with step four since step four uses the value in the ip register, however, step five can execute in parallel with any other step. Therefore, we can shave one cycle off the first two sequences above as follows:

#1 (step 4, step 5/6, and step 7)

#2 (step 4, step 5/7)

#3 (step 6 and step 7)

#4 (step 7)

Of course, there is no way to overlap the execution of steps seven and eight in the mov instruction since it must surely fetch the value before storing it away. By combining these steps, we obtain the following steps for the mov instruction:

- Fetch the instruction byte from memory.
- Decode the instruction and update ip
- If required, fetch a 16-bit instruction operand from memory.
- Compute the address of the operand, if required (i.e., bx+xxxx) .
- Fetch the operand, if required update ip to point beyond xxxx.

- Store the fetched value into the destination register

By adding a small amount of logic to the CPU, we've shaved one or two cycles off the execution of the mov instruction. This simple optimization works with most of the other instructions as well.

Another problem with the execution of the mov instruction concerns opcode alignment. Consider the mov ax, [1000] instruction that appears at location 100 in memory. The CPU spends one cycle fetching the opcode and, after decoding the instruction an determining it has a 16-bit operand, it takes two additional cycles to fetch that operand from memory (because that operand appears at an odd address – 101). The real travesty here is that the extra clock cycle to fetch these two bytes is unnecessary, after all, the CPU fetched the L.O. byte of the operand when it grabbed the opcode (remember, the x86 CPUs are 16-bit processors and always fetch 16 bits from memory), why not save that byte and use only one additional clock cycle to fetch the H.O. byte? This would shave one cycle off the execution time when the instruction begins at an even address (so the operand falls on an odd address). It would require only a one-byte register and a small amount of additional logic to accomplish this, well worth the effort.

While we are adding a register to buffer up operand bytes, let's consider some additional optimizations that could use the same logic. For example, consider what happens with that same mov instruction above executes. If we fetch the opcode and L.O. operand byte on the first cycle and the H.O. byte of the operand on the second cycle, we've actually read *four* bytes, not three. That fourth byte is the opcode of the next instruction. If we could save this opcode until the execution of the next instruction, we could shave a cycle of its execution time since it would not have to fetch the opcode byte. Furthermore, since the instruction decoder is idle while the CPU is executing the mov instruction, we can actually decode the next instruction while the current instruction is executing, thereby shaving yet another cycle off the execution of the next instruction. On the average, we will fetch this extra byte on every other instruction. Therefore, implementing this simple scheme will allow us to shave two cycles off about 50% of the instructions we execute.

Can we do anything about the other 50% of the instructions? The answer is yes. Note that the execution of the mov instruction is not accessing memory on every clock cycle. For example, while storing the data into the destination register the bus is idle. During time periods when the bus is idle we can *pre-fetch* instruction opcodes and operands and save these values for executing the next instruction.

The major improvement to the 8286 over the 886 processor is the *prefetch queue.* Whenever the CPU is not using the Bus Interface Unit (BIU), the BIU can fetch additional bytes from the instruction stream. Whenever the CPU needs an instruction or operand byte, it grabs the next available byte from the prefetch queue. Since the BIU grabs two bytes at a time from memory at one shot and the CPU generally consumes fewer than two bytes per clock cycle, any bytes the CPU would normally fetch from the instruction stream will already be sitting in the prefetch queue.

Note, however, that we're not guaranteed that all instructions and operands will be sitting in the prefetch queue when we need them. For example, the jmp 1000 instruction will invalidate the contents of the prefetch queue. If this instruction appears at location 400, 401, and 402 in memory, the prefetch queue will contain the bytes at addresses 403, 404, 405, 406, 407, etc. After loading ip with 1000 the bytes at addresses 403, etc., won't do us any good. So the system has to pause for a moment to fetch the double word at address 1000 before it can go on.

Another improvement we can make is to overlap instruction decoding with the last step of the previous instruction. After the CPU processes the operand, the next available byte in the prefetch queue is an opcode, and the CPU can decode it in anticipation of its execution. Of course, if the current instruction modifies the ip register, any time spent decoding the next instruction goes to waste, but since this occurs in parallel with other operations, it does not slow down the system.

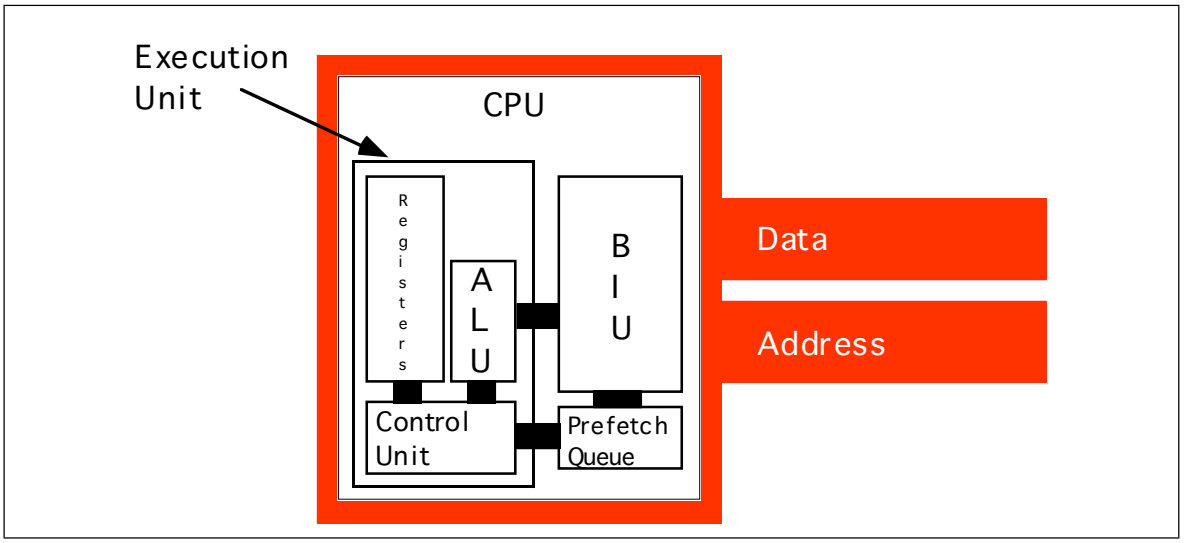


Figure 3.23 CPU With a Prefetch Queue

This sequence of optimizations to the system requires quite a few changes to the hardware. A block diagram of the system appears in Figure 3.23. The instruction execution sequence now assumes that the following events occur in the background:

CPU Prefetch Events:

- If the prefetch queue is not full (generally it can hold between eight and thirty-two bytes, depending on the processor) and the BIU is idle on the current clock cycle, fetch the next word from memory at the address in ip at the beginning of the clock cycle[14].
- If the instruction decoder is idle and the current instruction does not require an instruction operand, begin decoding the opcode at the front of the prefetch queue (if present), otherwise begin decoding the third byte in the prefetch queue (if present). If the desired byte is not in the prefetch queue, do not execute this event.

The instruction execution timings make a few optimistic assumptions, namely that any necessary opcodes and instruction operands are already present in the prefetch queue and that it has already decoded the current instruction opcode. If either cause is not true, an 8286 instruction's execution will delay while the system fetches the data from memory or decodes the instruction. The following are the steps for each of the 8286 instructions:

mov reg, mem/reg/const

- If required, compute the sum of [xxxx+bx] (1 cycle, if required).
- Fetch the source operand. Zero cycles if constant (assuming already in the prefetch queue), one cycle if a register, two cycles if even-aligned memory value, three cycles if odd-aligned memory value.
- Store the result in the destination register, one cycle.

mov mem, reg

- If required, compute the sum of [xxxx+bx] (1 cycle, if required).
- Fetch the source operand (a register), one cycle.
- Store into the destination operand. Two cycles if even-aligned memory value, three cycles if odd-aligned memory value.

instr reg, mem/reg/const (instr = add, sub, cmp, and, or)

- If required, compute the sum of [xxxx+bx] (1 cycle, if required).

14. This operation fetches only a byte if ip contains an odd value.

- Fetch the source operand. Zero cycles if constant (assuming already in the prefetch queue), one cycle if a register, two cycles if even-aligned memory value, three cycles if odd-aligned memory value.
- Fetch the value of the first operand (a register), one cycle.
- Compute the sum, difference, etc., as appropriate, one cycle.
- Store the result in the destination register, one cycle.

not mem/reg

- If required, compute the sum of [xxxx+bx] (1 cycle, if required).
- Fetch the source operand. One cycle if a register, two cycles if even-aligned memory value, three cycles if odd-aligned memory value.
- Logically not the value, one cycle.
- Store the result, one cycle if a register, two cycles if even-aligned memory value, three cycles if odd-aligned memory value.

jcc xxxx (conditional jump, cc=a, ae, b, be, e, ne)

- Test the current condition code (less than and equal) flags, one cycle.
- If the flag values are appropriate for the particular conditional branch, the CPU copies the 16-bit instruction operand into the ip register, one cycle.

jmp xxxx

- The CPU copies the 16-bit instruction operand into the ip register, one cycle.

As for the 886, we will not consider the execution times of the other x86 instructions since most of them are indeterminate.

The jump instructions look like they execute very quickly on the 8286. In fact, they may execute very slowly. Don't forget, jumping from one location to another invalidates the contents of the prefetch queue. So although the jmp instruction looks like it executes in one clock cycle, it forces the CPU to flush the prefetch queue and, therefore, spend several cycles fetching the next instruction, fetching additional operands, and decoding that instruction. Indeed, it make be two or three instructions after the jmp instruction before the CPU is back to the point where the prefetch queue is operating smoothly and the CPU is decoding opcodes in parallel with the execution of the previous instruction. The has one very important implication to your programs: *if you want to write fast code, make sure to avoid jumping around in your program as much as possible.*

Note that the conditional jump instructions only invalidate the prefetch queue if they actually make the jump. If the condition is false, they fall through to the next instruction and continue to use the values in the prefetch queue as well as any pre-decoded instruction opcodes. Therefore, if you can determine, while writing the program, which condition is most likely (e.g., less than vs. not less than), you should arrange your program so that the most common case falls through and conditional jump rather than take the branch.

Instruction size (in bytes) can also affect the performance of the prefetch queue. It never requires more than one clock cycle to fetch a single byte instruction, but it always requires two cycles to fetch a three-byte instruction. Therefore, if the target of a jump instruction is two one-byte instructions, the BIU can fetch both instructions in one clock cycle and begin decoding the second one while executing the first. If these instructions are three-byte instructions, the CPU may not have enough time to fetch and decode the second or third instruction by the time it finishes the first. Therefore, you should attempt to use shorter instructions whenever possible since they will improve the performance of the prefetch queue.

The following table provides the (optimistic) execution times for the 8286 instructions:

Table 20: Execution Times for 8286 Instructions

Instruction ⇒ Addressing Mode ⇓	mov (both forms)	add, sub, cmp, and, or,	not	jmp	jxx
reg, reg	2	4			
reg, xxxx	1	3			
reg, [bx]	3-4	5-6			
reg, [xxxx]	3-4	5-6			
reg, [xxxx+bx]	4-5	6-7			
[bx], reg	3-4	5-6			
[xxxx], reg	3-4	5-6			
[xxxx+bx], reg	4-5	6-7			
reg			3		
[bx]			5-7		
[xxxx]			5-7		
[xxxx+bx]			6-8		
xxxx				1+pfd[a]	2[b] 2+pfd

a. Cost of prefetch and decode on the next instruction.
b. If not taken.

Note how much faster the mov instruction runs on the 8286 compared to the 886. This is because the prefetch queue allows the processor to overlap the execution of adjacent instructions. However, this table paints an overly rosy picture. Note the disclaimer: "assuming the opcode is present in the prefetch queue and has been decoded." Consider the following three instruction sequence:

```
????:           jmp     1000
1000:           jmp     2000
2000:           mov     cx, 3000[bx]
```

The second and third instructions will *not* execute as fast as the timings suggest in the table above. Whenever we modify the value of the ip register the CPU flushes the prefetch queue. So the CPU cannot fetch and decode the next instruction. Instead, it must fetch the opcode, decode it, etc., increasing the execution time of these instructions. At this point the only improvement we've made is to execute the "update ip" operation in parallel with another step.

Usually, including the prefetch queue improves performance. That's why Intel provides the prefetch queue on every model of the 80x86, from the 8088 on up. On these processors, the BIU is constantly fetching data for the prefetch queue whenever the program is not actively reading or writing data.

Prefetch queues work best when you have a wide data bus. The 8286 processor runs much faster than the 886 because it can keep the prefetch queue full. However, consider the following instructions:

```
100:            mov     ax, [1000]
105:            mov     bx, [2000]
10A:            mov     cx, [3000]
```

Since the ax, bx, and cx registers are 16 bits, here's what happens (assuming the first instruction is in the prefetch queue and decoded):

- Fetch the opcode byte from the prefetch queue (zero cycles).
- Decode the instruction (zero cycles).
- There is an operand to this instruction, so get it from the prefetch queue (zero cycles).
- Get the value of the second operand (one cycle). Update ip.
- Store the fetched value into the destination register (one cycle). Fetch two bytes from code stream. Decode the next instruction.

End of first instruction. Two bytes currently in prefetch queue.

- Fetch the opcode byte from the prefetch queue (zero cycles).
- Decode the instruction to see what it does (zero cycles).
- If there is an operand to this instruction, get that operand from the prefetch queue (one clock cycle because we're still missing one byte).
- Get the value of the second operand (one cycle). Update ip.
- Store the fetched value into the destination register (one cycle). Fetch two bytes from code stream. Decode the next instruction.

End of second instruction. Three bytes currently in prefetch queue.

- Fetch the opcode byte from the prefetch queue (zero cycles).
- Decode the instruction (zero cycles).
- If there is an operand to this instruction, get that operand from the prefetch queue (zero cycles).
- Get the value of the second operand (one cycle). Update ip.
- Store the fetched value into the destination register (one cycle). Fetch two bytes from code stream. Decode the next instruction.

As you can see, the second instruction requires one more clock cycle than the other two instructions. This is because the BIU cannot fill the prefetch queue quite as fast as the CPU executes the instructions. This problem is exasperated when you limit the size of the prefetch queue to some number of bytes. This problem doesn't exist on the 8286 processor, but most certainly does exist in the 80x86 processors.

You'll soon see that the 80x86 processors tend to exhaust the prefetch queue quite easily. Of course, once the prefetch queue is empty, the CPU must wait for the BIU to fetch new opcodes from memory, slowing the program. Executing shorter instructions helps keep the prefetch queue full. For example, the 8286 can load *two* one-byte instructions with a single memory cycle, but it takes 1.5 clock cycles to fetch a single three-byte instruction. Usually, it takes longer to execute those four one-byte instructions than it does to execute the single three-byte instruction. This gives the prefetch queue time to fill and decode new instructions. In systems with a prefetch queue, it's possible to find eight two-byte instructions which operate faster than an equivalent set of four four-byte instructions. The reason is that the prefetch queue has time to refill itself with the shorter instructions.

Moral of the story: *when programming a processor with a prefetch queue, always use the shortest instructions possible to accomplish a given task.*

3.3.12 The 8486 Processor

Executing instructions in parallel using a bus interface unit and an execution unit is a special case of *pipelining*. The 8486 incorporates pipelining to improve performance. With just a few exceptions, we'll see that pipelining allows us to execute one instruction per clock cycle.

The advantage of the prefetch queue was that it let the CPU overlap instruction fetching and decoding with instruction execution. That is, while one instruction is executing, the BIU is fetching and decoding the next instruction. Assuming you're willing to add

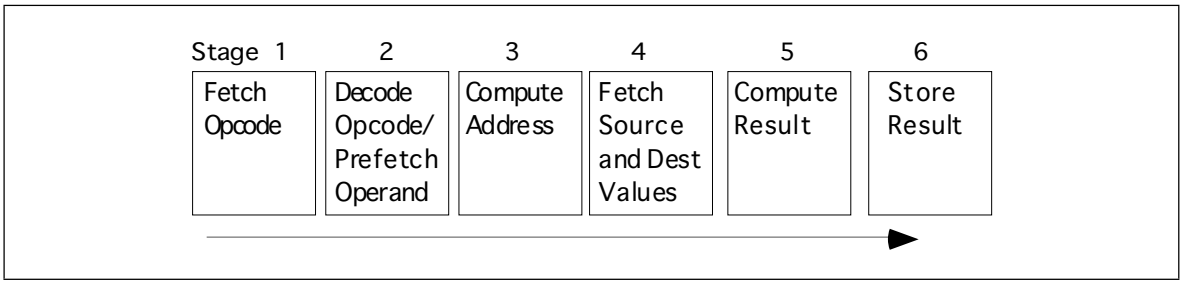


Figure 3.24 A Pipelined Implementation of Instruction Execution

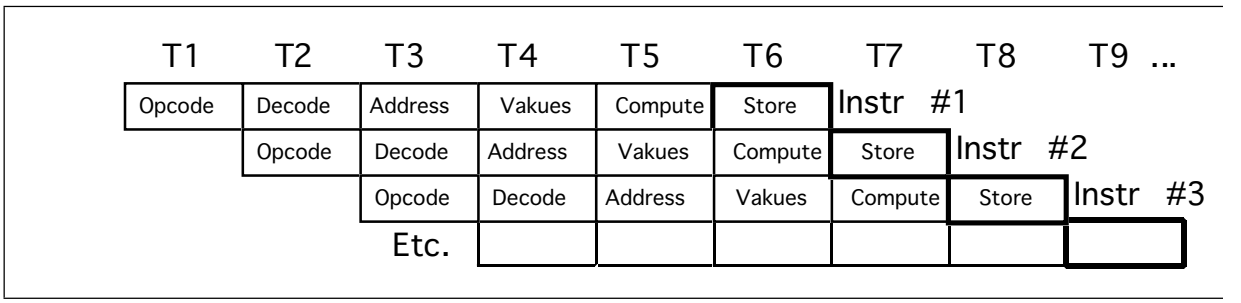


Figure 3.25 Instruction Execution in a Pipeline

hardware, you can execute almost all operations in parallel. That is the idea behind pipelining.

3.3.12.1 The 8486 Pipeline

Consider the steps necessary to do a generic operation:

- Fetch opcode.
- Decode opcode and (in parallel) prefetch a possible 16-bit operand.
- Compute complex addressing mode (e.g., [xxxx+bx]), if applicable.
- Fetch the source value from memory (if a memory operand) and the destination register value (if applicable).
- Compute the result.
- Store result into destination register.

Assuming you're willing to pay for some extra silicon, you can build a little "mini-processor" to handle each of the above steps. The organization would look something like Figure 3.24.

If you design a separate piece of hardware for each stage in the *pipeline* above, almost *all* these steps can take place in parallel. Of course, you cannot fetch and decode the opcode for any one instruction at the same time, but you can fetch one opcode while decoding the previous instruction. If you have an *n*-stage pipeline, you will usually have *n* instructions executing concurrently. The 8486 processor has a six stage pipeline, so it overlaps the execution of six separate instructions.

Figure 3.25, *Instruction Execution in a Pipeline*, demonstrates pipelining. T1, T2, T3, etc., represent consecutive "ticks" of the system clock. At T=T1 the CPU fetches the opcode byte for the first instruction.

At T=T2, the CPU begins decoding the opcode for the first instruction. In parallel, it fetches 16-bits from the prefetch queue in the event the instruction has an operand. Since the first instruction no longer needs the opcode fetching circuitry, the CPU instructs it to fetch the opcode of the second instruction in parallel with the decoding of the first instruction. Note there is a minor conflict here. The CPU is attempting to fetch the next byte from the prefetch queue for use as an operand, at the same time it is fetching 16 bits from the

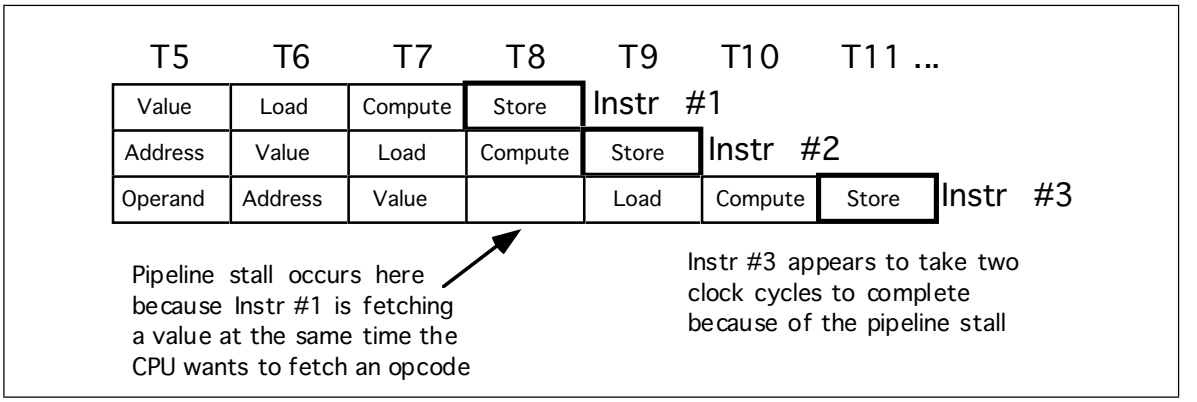


Figure 3.26 A Pipeline Stall

prefetch queue for use as an opcode. How can it do both at once? You'll see the solution in a few moments.

At T=T3 the CPU computes an operand address for the first instruction, if any. The CPU does nothing on the first instruction if it does not use the [xxxx+bx] addressing mode. During T3, the CPU also decodes the opcode of the second instruction and fetches any necessary operand. Finally the CPU also fetches the opcode for the third instruction. With each advancing tick of the clock, another step in the execution of each instruction in the pipeline completes, and the CPU fetches yet another instruction from memory.

At T=T6 the CPU completes the execution of the first instruction, computes the result for the second, etc., and, finally, fetches the opcode for the sixth instruction in the pipeline. The important thing to see is that after T=T5 the CPU completes an instruction on every clock cycle. *Once the CPU fills the pipeline, it completes one instruction on each cycle.* Note that this is true even if there are complex addressing modes to be computed, memory operands to fetch, or other operations which use cycles on a non-pipelined processor. All you need to do is add more stages to the pipeline, and you can still effectively process each instruction in one clock cycle.

3.3.12.2 Stalls in a Pipeline

Unfortunately, the scenario presented in the previous section is a little too simplistic. There are two drawbacks to that simple pipeline: bus contention among instructions and non-sequential program execution. Both problems may increase the average execution time of the instructions in the pipeline.

Bus contention occurs whenever an instruction needs to access some item in memory. For example, if a mov mem, reg instruction needs to store data in memory and a mov reg, mem instruction is reading data from memory, contention for the address and data bus may develop since the CPU will be trying to simultaneously fetch data and write data in memory.

One simplistic way to handle bus contention is through a *pipeline stall*. The CPU, when faced with contention for the bus, gives priority to the instruction furthest along in the pipeline. The CPU suspends fetching opcodes until the current instruction fetches (or stores) its operand. This causes the new instruction in the pipeline to take two cycles to execute rather than one (see Figure 3.26).

This example is but one case of bus contention. There are many others. For example, as noted earlier, fetching instruction operands requires access to the prefetch queue at the same time the CPU needs to fetch an opcode. Furthermore, on processors a little more advanced than the 8486 (e.g., the 80486) there are other sources of bus contention popping up as well. Given the simple scheme above, it's unlikely that most instructions would execute at one clock per instruction (CPI).

Fortunately, the intelligent use of a cache system can eliminate many pipeline stalls like the ones discussed above. The next section on caching will describe how this is done. However, it is not always possible, even with a cache, to avoid stalling the pipeline. What you cannot fix in hardware, you can take care of with software. If you avoid using memory, you can reduce bus contention and your programs will execute faster. Likewise, using shorter instructions also reduces bus contention and the possibility of a pipeline stall.

What happens when an instruction *modifies* the ip register? By the time the instruction

```
jmp     1000
```

completes execution, we've already started five other instructions and we're only one clock cycle away from the completion of the first of these. Obviously, the CPU must not execute those instructions or it will compute improper results.

The only reasonable solution is to *flush* the entire pipeline and begin fetching opcodes anew. However, doing so causes a severe execution time penalty. It will take six clock cycles (the length of the 8486 pipeline) before the next instruction completes execution. Clearly, you should avoid the use of instructions which interrupt the sequential execution of a program. This also shows another problem – pipeline length. The longer the pipeline is, the more you can accomplish per cycle in the system. However, lengthening a pipeline may slow a program if it jumps around quite a bit. Unfortunately, you cannot control the number of stages in the pipeline. You can, however, control the number of transfer instructions which appear in your programs. Obviously you should keep these to a minimum in a pipelined system.

3.3.12.3 Cache, the Prefetch Queue, and the 8486

System designers can resolve many problems with bus contention through the intelligent use of the prefetch queue and the cache memory subsystem. They can design the prefetch queue to buffer up data from the instruction stream, and they can design the cache with separate data and code areas. Both techniques can improve system performance by eliminating some conflicts for the bus.

The prefetch queue simply acts as a buffer between the instruction stream in memory and the opcode fetching circuitry. Unfortunately, the prefetch queue on the 8486 does not enjoy the advantage it had on the 8286. The prefetch queue works well for the 8286 because the CPU isn't constantly accessing memory. When the CPU isn't accessing memory, the BIU can fetch additional instruction opcodes for the prefetch queue. Alas, the 8486 CPU is constantly accessing memory since it fetches an opcode byte on every clock cycle. Therefore, the prefetch queue cannot take advantage of any "dead" bus cycles to fetch additional opcode bytes – there aren't any "dead" bus cycles. However, the prefetch queue is still valuable on the 8486 for a very simple reason: the BIU fetches two bytes on each memory access, yet some instructions are only one byte long. Without the prefetch queue, the system would have to explicitly fetch each opcode, even if the BIU had already "accidentally" fetched the opcode along with the previous instruction. With the prefetch queue, however, the system will not refetch any opcodes. It fetches them once and saves them for use by the opcode fetch unit.

For example, if you execute two one-byte instructions in a row, the BIU can fetch both opcodes in one memory cycle, freeing up the bus for other operations. The CPU can use these available bus cycles to fetch additional opcodes or to deal with other memory accesses.

Of course, not all instructions are one byte long. The 8486 has two instruction sizes: one byte and three bytes. If you execute several three-byte load instructions in a row, you're going to run slower, e.g.,

```
mov     ax, 1000
mov     bx, 2000
mov     cx, 3000
add     ax, 5000
```

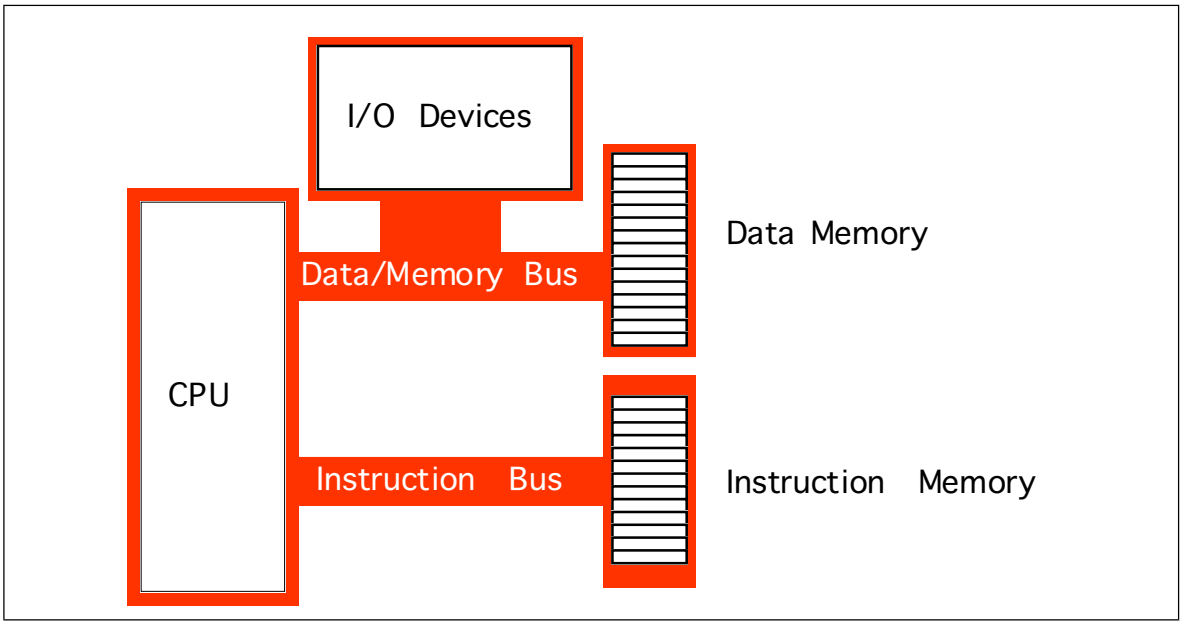


Figure 3.27 A Typical Harvard Machine

Each of these instructions reads an opcode byte and a 16 bit operand (the constant). Therefore, it takes an average of 1.5 clock cycles to read each instruction above. As a result, the instructions will require six clock cycles to execute rather than four.

Once again we return to that same rule: *the fastest programs are the ones which use the shortest instructions*. If you can use shorter instructions to accomplish some task, do so. The following instruction sequence provides a good example:

```
mov     ax, 1000
mov     bx, 1000
mov     cx, 1000
add     ax, 1000
```

We can reduce the size of this program and increase its execution speed by changing it to:

```
mov     ax, 1000
mov     bx, ax
mov     cx, ax
add     ax, ax
```

This code is only five bytes long compared to 12 bytes for the previous example. The previous code will take a minimum of five clock cycles to execute, more if there are other bus contention problems. The latter example takes only four[15]. Furthermore, the second example leaves the bus free for three of those four clock periods, so the BIU can load additional opcodes. Remember, *shorter* often means *faster*.

While the prefetch queue can free up bus cycles and eliminate bus contention, some problems still exist. Suppose the average instruction length for a sequence of instructions is 2.5 bytes (achieved by having three three-byte instructions and one one-byte instruction together). In such a case the bus will be kept busy fetching opcodes and instruction operands. There will be no free time left to access memory. Assuming some of those instructions access memory the pipeline will stall, slowing execution.

Suppose, for a moment, that the CPU has two separate memory spaces, one for instructions and one for data, each with their own bus. This is called the Harvard Architecture since the first such machine was built at Harvard. On a Harvard machine there would be no contention for the bus. The BIU could continue to fetch opcodes on the instruction bus while accessing memory on the data / memory bus (see Figure 3.27),

15. Actually, both of these examples will take longer to execute. See the section on hazards for more details.

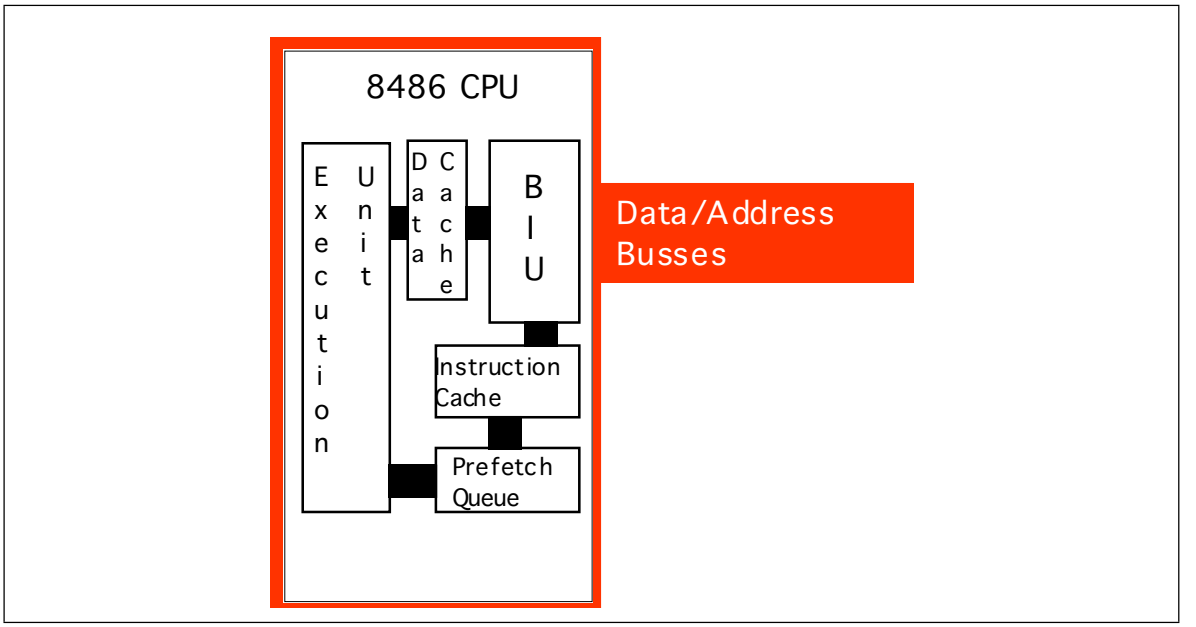


Figure 3.28 Internal Structure of the 8486 CPU

In the real world, there are very few true Harvard machines. The extra pins needed on the processor to support two physically separate busses increase the cost of the processor and introduce many other engineering problems. However, microprocessor designers have discovered that they can obtain many benefits of the Harvard architecture with few of the disadvantages by using separate on-chip caches for data and instructions. Advanced CPUs use an internal Harvard architecture and an external Von Neumann architecture. Figure 3.28 shows the structure of the 8486 with separate data and instruction caches.

Each path inside the CPU represents an independent bus. Data can flow on all paths concurrently. This means that the prefetch queue can be pulling instruction opcodes from the instruction cache while the execution unit is writing data to the data cache. Now the BIU only fetches opcodes from memory whenever it cannot locate them in the instruction cache. Likewise, the data cache buffers memory. The CPU uses the data/address bus only when reading a value which is not in the cache or when flushing data back to main memory.

By the way, the 8486 handles the instruction operand / opcode fetch contention problem in a sneaky fashion. By adding an extra decoder circuit, it decodes the instruction at the beginning of the prefetch queue and three bytes into the prefetch queue in parallel. Then, if the previous instruction did not have a 16-bit operand, the CPU uses the result from the first decoder; if the previous instruction uses the operand, the CPU uses the result from the second decoder.

Although you cannot control the presence, size, or type of cache on a CPU, as an assembly language programmer you must be aware of how the cache operates to write the best programs. On-chip instruction caches are generally quite small (8,192 bytes on the 80486, for example). Therefore, the shorter your instructions, the more of them will fit in the cache (getting tired of “shorter instructions” yet?). The more instructions you have in the cache, the less often bus contention will occur. Likewise, using registers to hold temporary results places less strain on the data cache so it doesn’t need to flush data to memory or retrieve data from memory quite so often. *Use the registers wherever possible!*

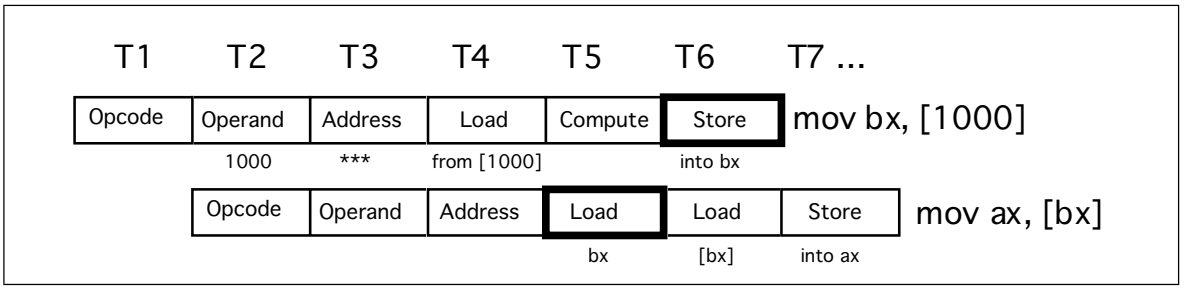


Figure 3.29 A Hazard on the 8486

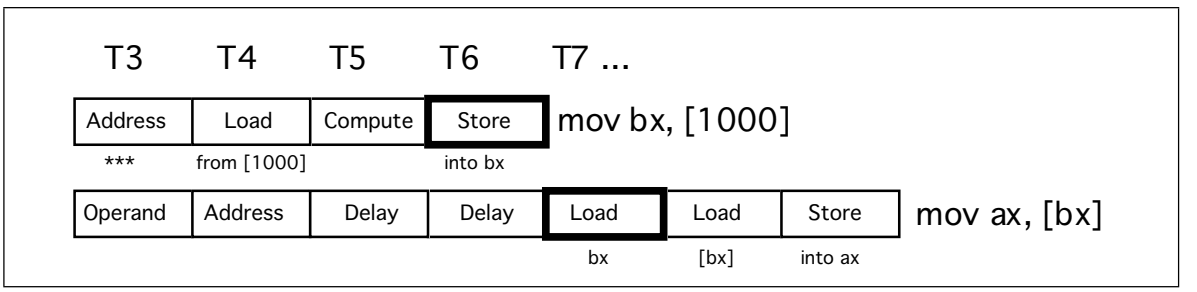


Figure 3.30 A Hazard on the 8486

3.3.12.4 Hazards on the 8486

There is another problem with using a pipeline: the data hazard. Let's look at the execution profile for the following instruction sequence:

```
mov     bx, [1000]
mov     ax, [bx]
```

When these two instructions execute, the pipeline will look something like Figure 3.29.

Note a major problem here. These two instructions fetch the 16 bit value whose address appears at location 1000 in memory. *But this sequence of instructions won't work properly!* Unfortunately, the second instruction has already used the value in bx before the first instruction loads the contents of memory location 1000 (T4 & T6 in the diagram above).

CISC processors, like the 80x86, handle hazards automatically[16]. However, they will stall the pipeline to synchronize the two instructions. The actual execution on the 8486 would look something like shown in Figure 3.29.

By delaying the second instruction two clock cycles, the 8486 guarantees that the load instruction will load ax from the proper address. Unfortunately, the second load instruction now executes in three clock cycles rather than one. However, requiring two extra clock cycles is better than producing incorrect results. Fortunately, you can reduce the impact of hazards on execution speed within your software.

Note that the data hazard occurs when the source operand of one instruction was a destination operand of a previous instruction. There is nothing wrong with loading bx from [1000] and then loading ax from [bx], *unless they occur one right after the other.* Suppose the code sequence had been:

```
mov     cx, 2000
mov     bx, [1000]
mov     ax, [bx]
```

16. RISC chips do not. If you tried this sequence on a RISC chip you would get an incorrect answer.

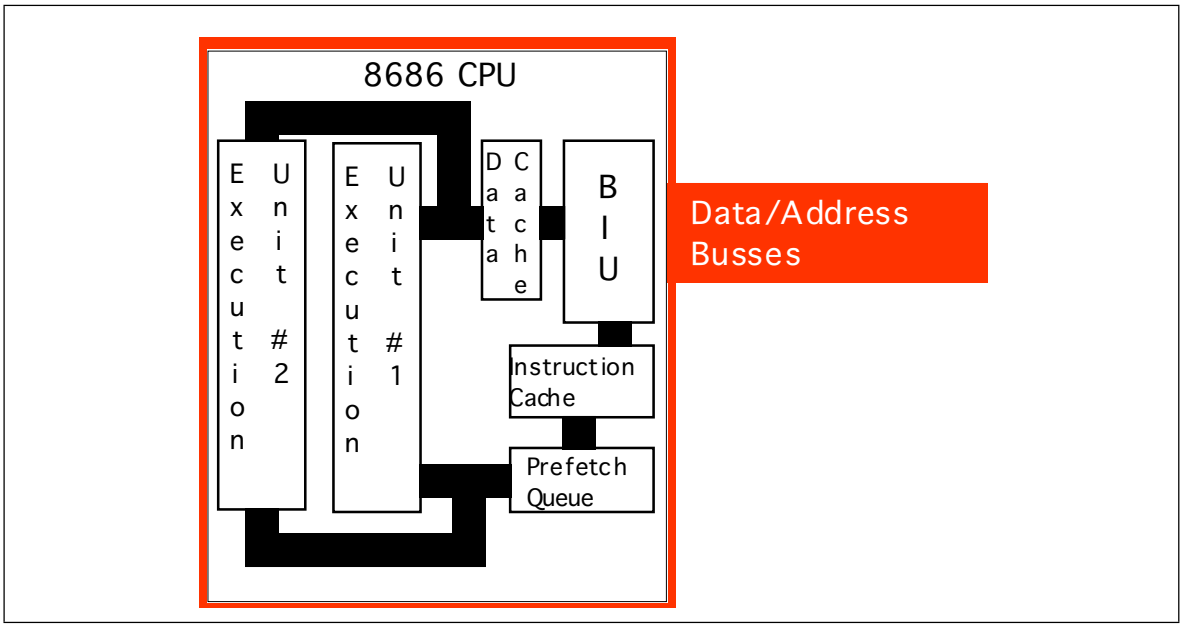


Figure 3.31 Internal Structure of the 8686 CPU

We could reduce the effect of the hazard that exists in this code sequence by simply *rearranging the instructions*. Let's do that and obtain the following:

```
mov     bx, [1000]
mov     cx, 2000
mov     ax, [bx]
```

Now the mov ax instruction requires only one additional clock cycle rather than two. By inserting yet another instruction between the mov bx and mov ax instructions you can eliminate the effects of the hazard altogether[17].

On a pipelined processor, the order of instructions in a program may dramatically affect the performance of that program. Always look for possible hazards in your instruction sequences. Eliminate them wherever possible by rearranging the instructions.

3.3.13 The 8686 Processor

With the pipelined architecture of the 8486 we could achieve, at best, execution times of one CPI (clock per instruction). Is it possible to execute instructions faster than this? At first glance you might think, "Of course not, we can do at most one operation per clock cycle. So there is no way we can execute more than one instruction per clock cycle." Keep in mind however, that a single instruction is *not* a single operation. In the examples presented earlier each instruction has taken between six and eight operations to complete. By adding seven or eight separate units to the CPU, we could effectively execute these eight operations in one clock cycle, yielding one CPI. If we add more hardware and execute, say, 16 operations at once, can we achieve 0.5 CPI? The answer is a qualified "yes." A CPU including this additional hardware is a *superscalar* CPU and can execute more than one instruction during a single clock cycle. That's the capability that the 8686 processor adds.

A superscalar CPU has, essentially, several execution units (see Figure 3.31). If it encounters two or more instructions in the instruction stream (i.e., the prefetch queue) which can execute independently, it will do so.

There are a couple of advantages to going superscalar. Suppose you have the following instructions in the instruction stream:

17. Of course, any instruction you insert at this point must *not* modify the values in the ax and bx registers.

```
mov     ax, 1000
mov     bx, 2000
```

If there are no other problems or hazards in the surrounding code, and all six bytes for these two instructions are currently in the prefetch queue, there is no reason why the CPU cannot fetch and execute both instructions in parallel. All it takes is extra silicon on the CPU chip to implement two execution units.

Besides speeding up independent instructions, a superscalar CPU can also speed up program sequences which have hazards. One limitation of the 8486 CPU is that once a hazard occurs, the offending instruction will completely stall the pipeline. Every instruction which follows will also have to wait for the CPU to synchronize the execution of the instructions. With a superscalar CPU, however, instructions following the hazard may continue execution through the pipeline as long as they don't have hazards of their own. This alleviates (though does not eliminate) some of the need for careful instruction scheduling.

As an assembly language programmer, the way you write software for a superscalar CPU can dramatically affect its performance. First and foremost is that rule you're probably sick of by now: *use short instructions*. The shorter your instructions are, the more instructions the CPU can fetch in a single operation and, therefore, the more likely the CPU will execute faster than one CPI. Most superscalar CPUs do not completely duplicate the execution unit. There might be multiple ALUs, floating point units, etc. This means that certain instruction sequences can execute very quickly while others won't. You have to study the exact composition of your CPU to decide which instruction sequences produce the best performance.

3.4 I/O (Input/Output)

There are three basic forms of input and output that a typical computer system will use: *I/O-mapped I/O*, *memory-mapped input/output*, and *direct memory access* (DMA). I/O-mapped input/output uses special instructions to transfer data between the computer system and the outside world; memory-mapped I/O uses special memory locations in the normal address space of the CPU to communicate with real-world devices; DMA is a special form of memory-mapped I/O where the peripheral device reads and writes memory without going through the CPU. Each I/O mechanism has its own set of advantages and disadvantages, we will discuss these in this section.

The first thing to learn about the input/output subsystem is that I/O in a typical computer system is radically different than I/O in a typical high level programming language. In a real computer system you will rarely find machine instructions that behave like writeln, printf, or even the x86 Get and Put instructions[18]. In fact, most input/output instructions behave exactly like the x86's mov instruction. To send data to an output device, the CPU simply moves that data to a special memory location (in the I/O address space if I/O-mapped input/output [see "The I/O Subsystem" on page 92] or to an address in the memory address space if using memory-mapped I/O). To read data from an input device, the CPU simply moves data from the address (I/O or memory) of that device into the CPU. Other than there are usually more wait states associated with a typical peripheral device than actual memory, the input or output operation looks very similar to a memory read or write operation (see "Memory Access and the System Clock" on page 93).

An I/O *port* is a device that looks like a memory cell to the computer but contains connections to the outside world. An I/O port typically uses a latch rather than a flip-flop to implement the memory cell. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system (see Figure 3.32). Note that I/O ports can be read-only, write-only, or read/write. The port in Figure 3.32, for example, is a write-only port. Since

18. Get and Put behave the way they do in order to simplify writing x86 programs.

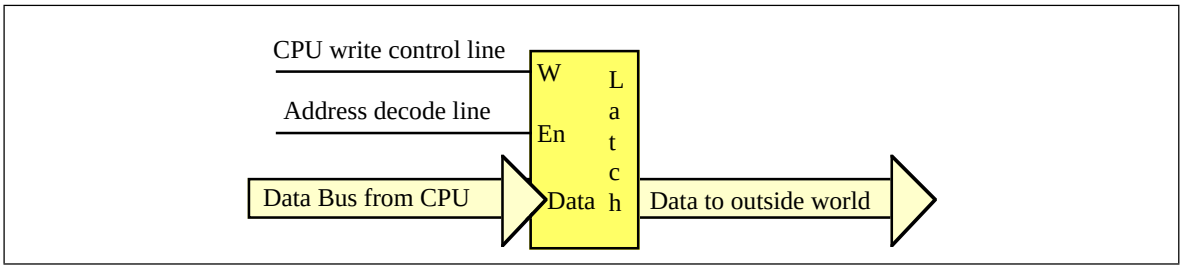


Figure 3.32 An Output Port Created with a Single Latch

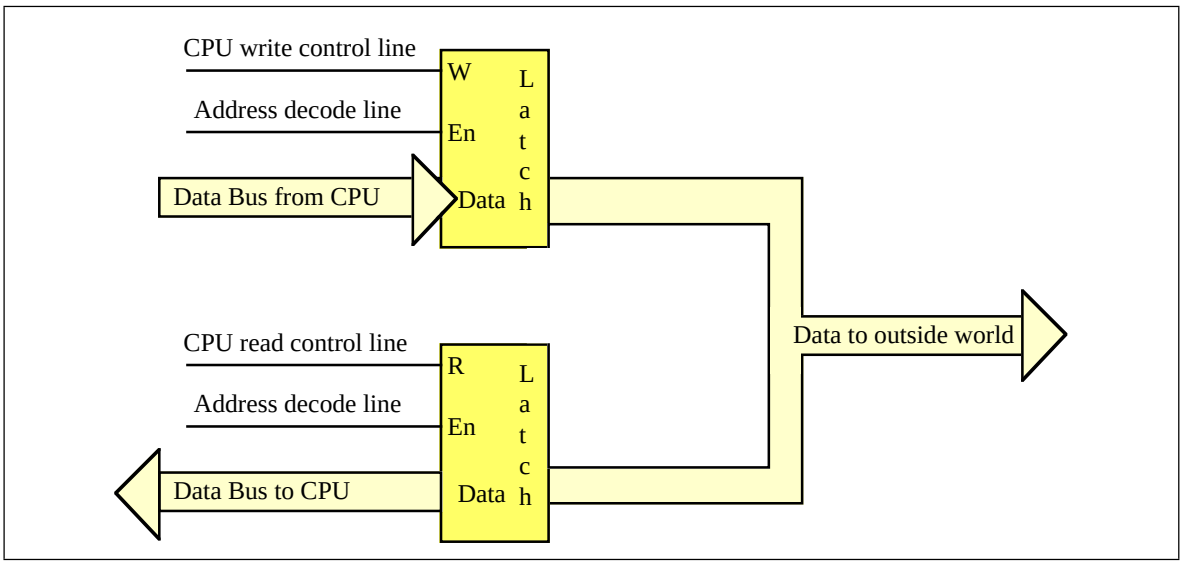


Figure 3.33 An Input/Output Port Requires Two Latches

the outputs on the latch do not loop back to the CPU's data bus, the CPU cannot read the data the latch contains. Both the address decode and write control lines must be active for the latch to operate; when reading from the latch's address the decode line is active, but the write control line is not.

Figure 3.33 shows how to create a read/write input/output port. The data written to the output port loops back to a transparent latch. Whenever the CPU reads the decoded address the read and decode lines are active and this activates the lower latch. This places the data previously written to the output port on the CPU's data bus, allowing the CPU to read that data. A read-only (input) port is simply the lower half of Figure 3.33; the system ignores any data written to an input port.

A perfect example of an output port is a parallel printer port. The CPU typically writes an ASCII character to a byte-wide output port that connects to the DB-25F connect on the back of the computer's case. A cable transmits this data to a the printer where an input port (to the printer) receives the data. A processor inside the printer typically converts this ASCII character to a sequence of dots it prints on the paper.

Generally, a given peripheral device will use more than a single I/O port. A typical PC parallel printer interface, for example, uses three ports: a read/write port, an input port, and an output port. The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port). The input port returns control signals from the printer; these signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc. The output port transmits control information to the printer such as whether data is available to print.

To the programmer, the difference between I/O-mapped and memory-mapped input/output operations is the instruction to use. For memory-mapped I/O, any instruction that accesses memory can access a memory-mapped I/O port. On the x86, the mov,

add, sub, cmp, and, or, and not instructions can read memory; the mov and not instructions can write data to memory. I/O-mapped input/output uses special instructions to access I/O ports. For example, the x86 CPUs use the get and put instructions[19], the Intel 80x86 family uses the in and out instructions. The 80x86 in and out instructions work just like the mov instruction except they place their address on the I/O address bus rather than the memory address bus (See "The I/O Subsystem" on page 92.).

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory. For example, to input a sequence of ten bytes from an input port and store these bytes into memory the CPU must read each value and store it into memory. For very high-speed I/O devices the CPU may be too slow when processing this data a byte at a time. Such devices generally contain an interface to the CPU's bus so it directly read and write memory. This is known as *direct memory access* since the peripheral device accesses memory directly, without using the CPU as an intermediary. This often allows the I/O operation to proceed in parallel with other CPU operations, thereby increasing the overall speed of the system. Note, however, that the CPU and DMA device cannot both use the address and data busses at the same time. Therefore, concurrent processing only occurs if the CPU has a cache and is executing code and accessing data found in the cache (so the bus is free). Nevertheless, even if the CPU must halt and wait for the DMA operation to complete, the I/O is still much faster since many of the bus operations during I/O or memory-mapped input/output consist of instruction fetches or I/O port accesses which are not present during DMA operations.

3.5 Interrupts and Polled I/O

Many I/O devices cannot accept data at an arbitrary rate. For example, a Pentium based PC is capable of sending several million characters a second to a printer, but that printer is (probably) unable to print that many characters each second. Likewise, an input device like a keyboard is unable to provide several million keystrokes per second (since it operates at human speeds, not computer speeds). The CPU needs some mechanism to coordinate data transfer between the computer system and its peripheral devices.

One common way to coordinate data transfer is to provide some *status bits* in a secondary input port. For example, a one in a single bit in an I/O port can tell the CPU that a printer is ready to accept more data, a zero would indicate that the printer is busy and the CPU should not send new data to the printer. Likewise, a one bit in a different port could tell the CPU that a keystroke from the keyboard is available at the keyboard data port, a zero in that same bit could indicate that no keystroke is available. The CPU can test these bits prior to reading a key from the keyboard or writing a character to the printer.

Assume that the printer data port is memory-mapped to address 0FFE0h and the printer status port is bit zero of memory-mapped port 0FFE2h. The following code waits until the printer is ready to accept a byte of data and then it writes the byte in the L.O. byte of ax to the printer port:

```
0000:  mov      bx, [FFE2]
0003:  and      bx, 1
0006:  cmp      bx, 0
0009:  je       0000
000C:  mov      [FFE0], ax
         .        .
         .        .
         .        .
```

The first instruction fetches the data at the status input port. The second instruction logically ands this value with one to clear bits one through fifteen and set bit zero to the current status of the printer port. Note that this produces the value zero in bx if the printer

19. Get and put are a little fancier than true I/O-mapped instructions, but we will ignore that difference here.

is busy, it produces the value one in bx if the printer is ready to accept additional data. The third instruction checks bx to see if it contains zero (i.e., the printer is busy). If the printer is busy, this program jumps back to location zero and repeats this process over and over again until the printer status bit is one[20].

The following code provides an example of reading a keyboard. It presumes that the keyboard status bit is bit zero of address 0FFE6h (zero means no key pressed) and the ASCII code of the key appears at address 0FFE4h when bit zero of location 0FFE6h contains a one:

```
0000:  mov      bx, [FFE6]
0003:  and      bx, 1
0006:  cmp      bx, 0
0009:  je       0000
000C:  mov      ax, [FFE4]
        .        .
        .        .
        .        .
```

This type of I/O operation, where the CPU constantly tests a port to see if data is available, is *polling*, that is, the CPU polls (asks) the port if it has data available or if it is capable of accepting data. Polled I/O is inherently inefficient. Consider what happens in the previous code segment if the user takes ten seconds to press a key on the keyboard – the CPU spins in a loop doing nothing (other than testing the keyboard status port) for those ten seconds.

In early personal computer systems (e.g., the Apple II), this is exactly how a program would read data from the keyboard; when it wanted to read a key from the keyboard it would poll the keyboard status port until a key was available. Such computers could not do other operations while waiting for keystrokes. More importantly, if too much time passes between checking the keyboard status port, the user could press a second key and the first keystroke would be lost[21].

The solution to this problem is to provide an *interrupt* mechanism. An interrupt is an external hardware event (like a keypress) that causes the CPU to interrupt the current instruction sequence and call a special *interrupt service routine*. (ISR). An interrupt service routine typically saves all the registers and flags (so that it doesn't disturb the computation it interrupts), does whatever operation is necessary to handle the source of the interrupt, it restores the registers and flags, and then it resumes execution of the code it interrupted. In many computer systems (e.g., the PC), many I/O devices generate an interrupt whenever they have data available or are able to accept data from the CPU. The ISR quickly processes the request in the *background*, allowing some other computation to proceed normally in the *foreground*.

CPUs that support interrupts must provide some mechanism that allows the programmer to specify the address of the ISR to execute when an interrupt occurs. Typically, an *interrupt vector* is a special memory location that contains the address of the ISR to execute when an interrupt occurs. The x86 CPUs, for example, contain two interrupt vectors: one for a general purpose interrupt and one for a *reset* interrupt (the reset interrupt corresponds to pressing the reset button on most PCs). The Intel 80x86 family supports up to 256 different interrupt vectors.

After an ISR completes its operation, it generally returns control to the foreground task with a special "return from interrupt" instruction. On the x86 the iret (interrupt return) instruction handles this task. An ISR should always end with this instruction so the ISR can return control to the program it interrupted.

A typical interrupt-driven input system uses the ISR to read data from an input port and buffer it up whenever data becomes available. The foreground program can read that

20. Note that this is a hypothetical example. The PC's parallel printer port is *not* mapped to memory addresses 0FFE0h and 0FFE2h on the x86.

21. A keyboard data port generally provides only the last character typed, it does not provide a "keyboard buffer" for the system.

data from the buffer at its leisure without losing any data from the port. Likewise, a typical interrupt-driven output system (that gets an interrupt whenever the output device is ready to accept more data) can remove data from a buffer whenever the peripheral device is ready to accept new data.

3.6 Laboratory Exercises

In this laboratory you will use the "SIMX86.EXE" program found in the Chapter Three subdirectory. This program contains a built-in assembler (compiler), debugger, and interrupter for the x86 hypothetical CPUs. You will learn how to write basic x86 assembly language programs, assemble (compile) them, modify the contents of memory, and execute your x86 programs. You will also experiment with memory-mapped I/O, I/O-mapped input/output, DMA, and polled as well as interrupt-driven I/O systems.

In this set of laboratory exercises you will use the SIMx86.EXE program to enter, edit, initialize, and emulate x86 programs. This program requires that you install two files in your WINDOWS\SYSTEM directory. Please see the README.TXT file in the CH3 subdirectory for more details.

3.6.1 The SIMx86 Program – Some Simple x86 Programs

To run the SIMx86 program double click on its icon or choose run from the Windows file menu and enter the pathname for SIMx86. The SIMx86 program consists of three main screen that you can select by clicking on the *Editor, Memory, or Emulator* notebook tabs in the window. By default, SIMx86 opens the Editor screen. From the Editor screen you can edit and assemble x86 programs; from Memory screen you can view and modify the contents of memory; from the Emulator screen you execute x86 programs and view x86 programs in memory.

The SIMx86 program contains two menu items: File and Edit. These are standard Windows menus so there is little need to describe their operation except for two points. First, the New, Open, Save, and Save As items under the file menu manipulate the data in the text editor box on the Editor screen, they do not affect anything on the other screens. Second, the Print menu item in the File menu prints the source code appearing in the text editor if the Editor screen is active, it prints the entire form if the Memory or Emulator screens are active.

To see how the SIMx86 program operates, switch to the Editor screen (if you are not already there). Select "Open" from the File menu and choose "EX1.X86" from the Chapter Three subdirectory. That file should look like the following:

```
mov     ax, [1000]
mov     bx, [1002]
add     ax, bx
sub     ax, 1
mov     bx, ax
add     bx, ax
add     ax, bx
halt
```

This short code sequence adds the two values at location 1000 and 1002, subtracts one from their sum, and multiplies the result by three ((ax + ax) + ax) = ax*3), leaving the result in ax and then it halts.

On the Editor screen you will see three objects: the editor window itself, a box that holds the "Starting Address," and an "Assemble" button. The "Starting Address" box holds a hexadecimal number that specifies where the assembler will store the machine code for the x86 program you write with the editor. By default, this address is zero. About the only time you should change this is when writing interrupt service routines since the default reset address is zero. The "Assemble" button directs the SIMx86 program to con-

vert your assembly language source code into x86 machine code and store the result beginning at the Starting Address in memory. Go ahead and press the "Assemble" button at this time to assemble this program to memory.

Now press the "Memory" tab to select the memory screen. On this screen you will see a set of 64 boxes arranged as eight rows of eight boxes. To the left of these eight rows you will see a set of eight (hexadecimal) memory addresses (by default, these are 0000, 0008, 0010, 0018, 0020, 0028, 0030, and 0038). This tells you that the first eight boxes at the top of the screen correspond to memory locations 0, 1, 2, 3, 4, 5, 6, and 7; the second row of eight boxes correspond to locations 8, 9, A, B, C, D, E, and F; and so on. At this point you should be able to see the machine codes for the program you just assembled in memory locations 0000 through 000D. The rest of memory will contain zeros.

The memory screen lets you look at and possibly modify 64 bytes of the total 64K memory provided for the x86 processors. If you want to look at some memory locations other than 0000 through 003F, all you need do is edit the first address (the one that currently contains zero). At this time you should change the starting address of the memory display to 1000 so you can modify the values at addresses 1000 and 1002 (remember, the program adds these two values together). Type the following values into the corresponding cells: at address 1000 enter the value 34, at location 1001 the value 12, at location 1002 the value 01, and at location 1003 the value 02. Note that if you type an illegal hexadecimal value, the system will turn that cell red and beep at you.

By typing an address in the memory display starting address cell, you can look at or modify locations almost anywhere in memory. Note that if you enter a hexadecimal address that is not an even multiple of eight, the SIMx86 program disable up to seven cells on the first row. For example, if you enter the starting address 1002, SIMx86 will disable the first two cells since they correspond to addresses 1000 and 1001. The first active cell is 1002. Note the SIMx86 reserves memory locations FFF0 through FFFF for memory-mapped I/O. Therefore, it will not allow you to edit these locations. Addresses FFF0 through FFF7 correspond to read-only input ports (and you will be able to see the input values even though SIMx86 disables these cells). Locations FFF8 through FFFF are write-only output ports, SIMx86 displays garbage values if you look at these locations.

On the Memory page along with the memory value display/edit cells, there are two other entry cells and a button. The "Clear Memory" button clears memory by writing zeros throughout. Since your program's object code and initial values are currently in memory, you should not press this button. If you do, you will need to reassemble your code and reenter the values for locations 1000 through 1003.

The other two items on the Memory screen let you set the interrupt vector address and the reset vector address. By default, the reset vector address contains zero. This means that the SIMx86 begins program execution at address zero whenever you reset the emulator. Since your program is currently sitting at location zero in memory, you should not change the default reset address.

The "Interrupt Vector" value is FFFF by default. FFFF is a special value that tells SIMx86 "there is no interrupt service routine present in the system, so ignore all interrupts." Any other value must be the address of an ISR that SIMx86 will call whenever an interrupt occurs. Since the program you assembled does not have an interrupt service routine, you should leave the interrupt vector cell containing the value FFFF.

Finally, press the "Emulator" tab to look at the emulator screen. This screen is much busier than the other two. In the upper left hand corner of the screen is a data entry box with the label IP. This box holds the current value of the x86 *instruction pointer* register. Whenever SIMx86 runs a program, it begins execution with the instruction at this address. Whenever you press the reset button (or enter SIMx86 for the first time), the IP register contains the value found in the reset vector. If this register does not contain zero at this point, press the reset button on the Emulator screen to reset the system.

Immediately below the ip value, the Emulator page *disassembles* the instruction found at the address in the ip register. This is the very next instruction that SIMx86 will execute when you press the "Run" or "Step" buttons. Note that SIMx86 does not obtain this

instruction from the source code window on the Editor screen. Instead, it decodes the opcode in memory (at the address found in ip) and generates this string itself. Therefore, there may be minor differences between the instruction you wrote and the instruction SIMx86 displays on this page. Note that a disassembled instruction contains several numeric values in front of the actual instruction. The first (four-digit) value is the memory address of that instruction. The next pair of digits (or the next three pairs of digits) are the opcodes and possible instruction operand values. For example, the mov ax, [1000] instruction's machine code is C6 00 10 since these are the three sets of digits appearing at this point.

Below the current disassembled instruction, SIMx86 displays 15 instructions it disassembles. The starting address for this disassemble is *not* the value in the ip register. Instead, the value in the lower right hand corner of the screen specifies the starting disassembly address. The two little arrows next to the disassembly starting address let you quickly increment or decrement the disassembly starting address. Assuming the starting address is zero (change it to zero if it is not), press the down arrow. Note that this increments the starting address by one. Now look back at the disassembled listing. As you can see, pressing the down arrow has produced an interesting result. The first instruction (at address 0001) is "****". The four asterisks indicate that this particular opcode is an illegal instruction opcode. The second instruction, at address 0002, is not ax. Since the program you assembled did not contain an illegal opcode or a not ax instruction, you may be wondering where these instructions came from. However, note the starting address of the first instruction: 0001. This is the second byte of the first instruction in your program. In fact, the illegal instruction (opcode=00) and the not ax instruction (opcode=10) are actually a disassembly of the mov ax, [1000] two-byte operand. This should clearly demonstrate a problem with disassembly – it is possible to get "out of phase" by specify a starting address that is in the middle of a multi-byte instruction. You will need to consider this when disassembling code.

In the middle of the Emulator screen there are several buttons: Run, Step, Halt, Interrupt, and Reset (the "Running" box is an annunciator, not a button). Pressing the Run button will cause the SIMx86 program to run the program (starting at the address in the ip register) at "full" speed. Pressing the Step button instructs SIMx86 to execute only the instruction that ip points at and then stop. The Halt button, which is only active while a program is running, will stop execution. Pressing the Interrupt button generates an interrupt and pressing the Reset button resets the system (and halts execution if a program is currently running). Note that pressing the Reset button clears the x86 registers to zero and loads the ip register with the value in the reset vector.

The "Running" annunciator is gray if SIMx86 is not currently running a program. It turns red when a program is actually running. You can use this annunciator as an easy way to tell if a program is running if the program is busy computing something (or is in an infinite loop) and there is no I/O to indicate program execution.

The boxes with the ax, bx, cx, and dx labels let you modify the values of these registers while a program is not running (the entry cells are not enabled while a program is actually running). These cells also display the current values of the registers whenever a program stops or between instructions when you are stepping through a program. Note that while a program is running the values in these cells are static and do not reflect their current values.

The "Less" and "Equal" check boxes denote the values of the less than and equal flags. The x86 cmp instruction sets these flags depending on the result of the comparison. You can view these values while the program is not running. You can also initialize them to true or false by clicking on the appropriate box with the mouse (while the program is not running).

In the middle section of the Emulator screen there are four "LEDs" and four "toggle switches." Above each of these objects is a hexadecimal address denoting their memory-mapped I/O addresses. Writing a zero to a corresponding LED address turns that LED "off" (turns it white). Writing a one to a corresponding LED address turns that LED

"on" (turns it red). Note that the LEDs only respond to bit zero of their port addresses. These output devices ignore all other bits in the value written to these addresses.

The toggle switches provide four memory-mapped input devices. If you read the address above each switch SIMx86 will return a zero if the switch is off. SIMx86 will return a one if the switch is in the on position. You can toggle a switch by clicking on it with the mouse. Note that a little rectangle on the switch turns red if the switch is in the "on" position.

The two columns on the right side of the Emulate screen ("Input" and "Output") display input values read with the get instruction and output values the put instruction prints.

For this first exercise, you will use the Step button to single step through each of the instructions in the EX1.x86 program. First, begin by pressing the Reset button[22]. Next, press the Step button once. Note that the values in the ip and ax registers change. The ip register value changes to 0003 since that is the address of the next instruction in memory, ax's value changed to 1234 since that's the value you placed at location 1000 when operating on the Memory screen. Single step through the remaining instructions (by repeatedly pressing Step) until you get the "Halt Encountered" dialog box.

For your lab report: explain the results obtained after the execution of each instruction. Note that single-stepping through a program as you've done here is an excellent way to ensure that you fully understand how the program operates. As a general rule, you should always single-step through every program you write when testing it.

3.6.2 Simple I/O-Mapped Input/Output Operations

Go to the Editor screen and load the EX2.x86 file into the editor. This program introduces some new concepts, so take a moment to study this code:

```
                mov     bx, 1000
a:              get
                mov     [bx], ax
                add     bx, 2
                cmp     ax, 0
                jne     a

                mov     cx, bx
                mov     bx, 1000
                mov     ax, 0
b:              add     ax, [bx]
                add     bx, 2
                cmp     bx, cx
                jb      b

                put
                halt
```

The first thing to note are the two strings "a:" and "b:" appearing in column one of the listing. The SIMx86 assembler lets you specify up to 26 statement *labels* by specifying a single alphabetic character followed by a colon. Labels are generally the operand of a jump instruction of some sort. Therefore, the "jne a" instruction above really says "jump if not equal to the statement prefaced with the 'a:' label" rather than saying "jump if not equal to location ten (0Ah) in memory."

Using labels is much more convenient than figuring out the address of a target instruction manually, especially if the target instruction appears later in the code. The SIMx86 assembler computes the address of these labels and substitutes the correct address

22. It is a good idea to get in the habit of pressing the Reset button before running or stepping through any program.

for the operands of the jump instructions. Note that you *can* specify a numeric address in the operand field of a jump instruction. However, all numeric addresses must begin with a decimal digit (even though they are hexadecimal values). If your target address would normally begin with a value in the range A through F, simply prepend a zero to the number. For example, if "jne a" was supposed to mean "jump if not equal to location 0Ah" you would write the instruction as "jne 0a".

This program contains two loops. In the first loop, the program reads a sequence of values from the user until the user enters the value zero. This loop stores each word into successive memory locations starting at address 1000h. Remember, each word read by the user requires two bytes; this is why the loop adds two to bx on each iteration.

The second loop in this program scans through the input values and computes their sum. At the end of the loop, the code prints the sum to the output window using the put instruction.

For your lab report: single-step through this program and describe how each instruction works. Reset the x86 and run this program at full speed. Enter several values and describe the result. Discuss the get and put instruction. Describe why they do I/O-mapped input/output operations rather than memory-mapped input/output operations.

3.6.3 Memory Mapped I/O

From the Editor screen, load the EX3.x86 program file. That program takes the following form (the comments were added here to make the operation of this program clearer):

```
a:              mov     ax, [fff0]
                mov     bx, [fff2]

                mov     cx, ax      ;Computes Sw0 and Sw1
                and     cx, bx
                mov     [fff8], cx

                mov     cx, ax      ;Computes Sw0 or Sw1
                or      cx, bx
                mov     [fffa], cx

                mov     cx, ax      ;Computes Sw0 xor Sw1
                mov     dx, bx      ;Remember, xor = AB' + A'B
                not     cx
                not     bx
                and     cx, bx
                and     dx, ax
                or      cx, dx
                mov     [fffc], cx

                not     cx          ;Computes Sw0 = Sw1
                mov     [fffe], cx  ;Remember, equals = not xor

                mov     ax, [fff4]  ;Read the third switch.
                cmp     ax, 0       ;See if it's on.
                je      a           ;Repeat this program while off.
                halt
```

Locations 0FFF0h, 0FFF2h, and 0FFF4h correspond to the first three toggle switches on the Execution page. These are memory-mapped I/O devices that put a zero or one into the corresponding memory locations depending upon whether the toggle switch is in the on or off state. Locations 0FFF8h, 0FFFAh, 0FFFCh, and 0FFFEh correspond to the four LEDs. Writing a zero to these locations turns the corresponding LED off, writing a one turns it on.

This program computes the logical and, or, xor, and xnor (not xor) functions for the values read from the first two toggle switches. This program displays the results of these functions on the four output LEDs. This program reads the value of the third toggle switch to determine when to quit. When the third toggle switch is in the on position, the program will stop.

For your lab report: run this program and cycle through the four possible combinations of on and off for the first two switches. Include the results in your lab report.

3.6.4 DMA Exercises

In this exercise you will start a program running (EX4.x86) that examines and operates on values found in memory. Then you will switch to the Memory screen and modify values in memory (that is, you will directly access memory while the program continues to run), thus simulating a peripheral device that uses DMA.

The EX4.x86 program begins by setting memory location 1000h to zero. Then it loops until one of two conditions is met – either the user toggles the FFF0 switch or the user changes the value in memory location 1000h. Toggling the FFF0 switch terminates the program. Changing the value in memory location 1000h transfers control to a section of the program that adds together *n* words, where *n* is the new value in memory location 1000h. The program sums the words appearing in contiguous memory locations starting at address 1002h. The actual program looks like the following:

```
d:      mov       cx, 0                ;Clear location 1000h before we
        mov       [1000], cx           ; begin testing it.

; The following loop checks to see if memory location 1000h changes or if
; the FFF0 switch is in the on position.

a:      mov       cx, [1000]           ;Check to see if location 1000h
        cmp       cx, 0                ; changes. Jump to the section that
        jne       c                    ; sums the values if it does.

        mov       ax, [fff0]           ;If location 1000h still contains zero,
        cmp       ax, 0                ; read the FFF0 switch and see if it is
        je        a                    ; off. If so, loop back. If the switch
        halt                           ; is on, quit the program.

; The following code sums up the “cx” contiguous words of memory starting at
; memory location 1002. After it sums up these values, it prints their sum.

c:      mov       bx, 1002             ;Initialize BX to point at data array.
        mov       ax, 0                ;Initialize the sum
b:      add       ax, [bx]             ;Sum in the next array value.
        add       bx, 2                ;Point BX at the next item in the array.
        sub       cx, 1                ;Decrement the element count.
        cmp       cx, 0                ;Test to see if we’ve added up all the
        jne       b                    ; values in the array.

        put                            ;Print the sum and start over.
        jmp       d
```

Load this program into SIMx86 and assemble it. Switch to the Emulate screen, press the Reset button, make sure the FFF0 switch is in the off position, and then run the program. Once the program is running switch to the memory screen by pressing the Memory tab. Change the starting display address to 1000. Change the value at location 1000h to 5. Switch back to the emulator screen. Assuming memory locations 1002 through 100B all contain zero, the program should display a zero in the output column.

Switch back to the memory page. What does location 1000h now contain? Change the L.O. bytes of the words at address 1002, 1004, and 1006 to 1, 2, and 3, respectively. Change

the value in location 1000h to three. Switch to the Emulator page. Describe the output in your lab report. Try entering other values into memory. Toggle the FFF0 switch when you want to quit running this program.

For your lab report: explain how this program uses DMA to provide program input. Run several tests with different values in location 1000h and different values in the data array starting at location 1002. Include the results in your report.

For additional credit: Store the value 12 into memory location 1000. Explain why the program prints *two* values instead of just one value.

3.6.5 Interrupt Driven I/O Exercises

In this exercise you will load *two* programs into memory: a main program and an interrupt service routine. This exercise demonstrates the use of interrupts and an interrupt service routine.

The main program (EX5a.x86) will constantly compare memory locations 1000h and 1002h. If they are not equal, the main program will print the value of location 1000h and then copy this value to location 1002h and repeat this process. The main program repeats this loop until the user toggles switch FFF0 to the on position. The code for the main program is the following:

```
a:      mov       ax, [1000]          ;Fetch the data at location 1000h and
        cmp       ax, [1002]          ; see if it is the same as location
        je        b                   ; 1002h. If so, check the FFF0 switch.
        put                           ;If the two values are different, print
        mov       [1002], ax          ; 1000h's value and make them the same.

b:      mov       ax, [fff0]          ;Test the FFF0 switch to see if we
        cmp       ax, 0               ; should quit this program.
        je        a
        halt
```

The interrupt service routine (EX5b.x86) sits at location 100h in memory. Whenever an interrupt occurs, this ISR simply increments the value at location 1000h by loading this value into ax, adding one to the value in ax, and then storing this value back to location 1000h. After these instructions, the ISR returns to the main program. The interrupt service routine contains the following code:

```
        mov       ax, [1000]          ;Increment location 1000h by one and
        add       ax, 1               ; return to the interrupted code.
        mov       [1000], ax
        iret
```

You must load and assemble both files before attempting to run the main program. Begin by loading the main program (EX5a.x86) into memory and assemble it at address zero. Then load the ISR (EX5b.x86) into memory, set the Starting Address to 100, and then assemble your code. **Warning:** *if you forget to change the starting address you will wipe out your main program when you assemble the ISR. If this happens, you will need to repeat this procedure from the beginning.*

After assembling the code, the next step is to set the interrupt vector so that it contains the address of the ISR. To do this, switch to the Memory screen. The interrupt vector cell should currently contain 0FFFFh (this value indicates that interrupts are disabled). Change this to 100 so that it contains the address of the interrupt service routine. This also enables the interrupt system.

Finally, switch to the Emulator screen, make sure the FFF0 toggle switch is in the off position, reset the program, and start it running. Normally, nothing will happen. Now press the interrupt button and observe the results.

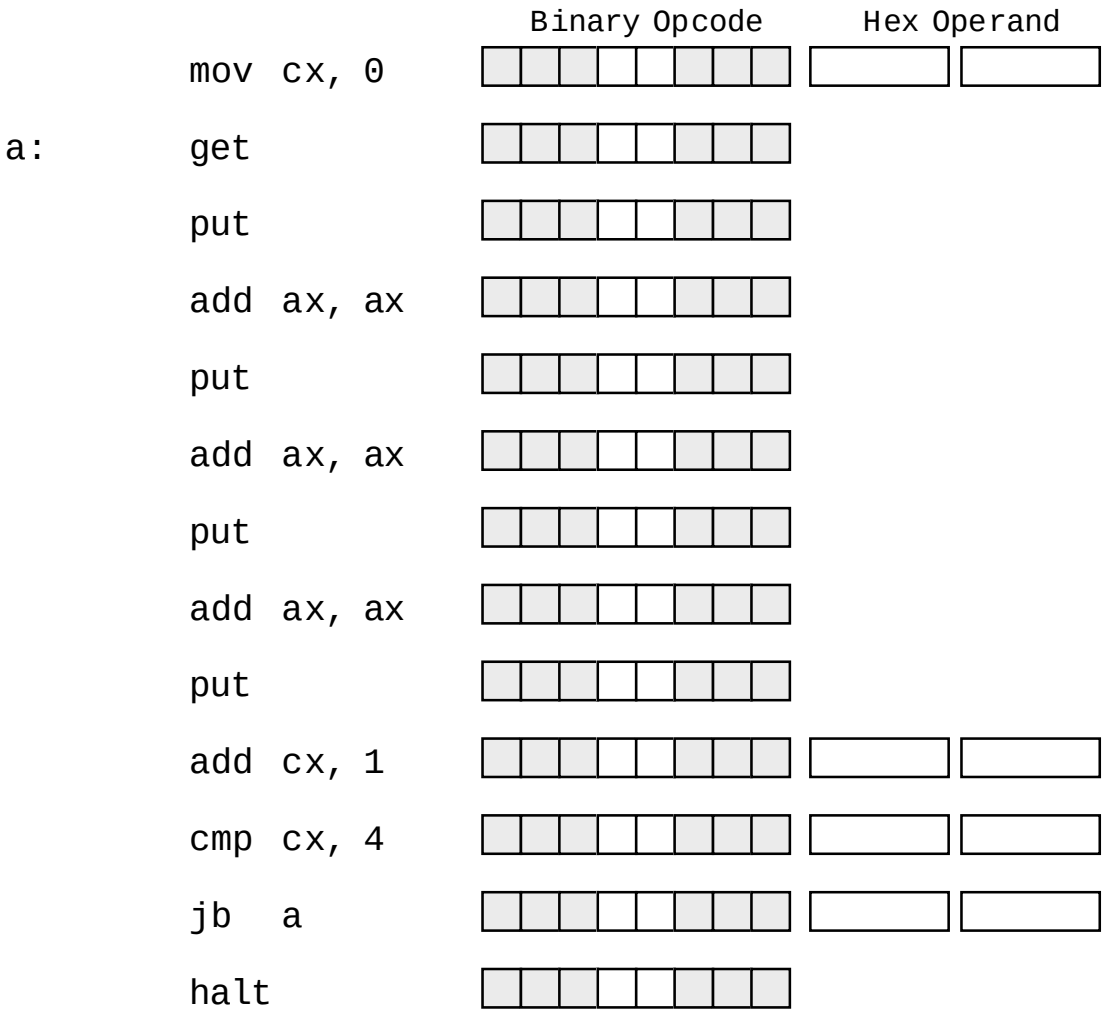
For your lab report: describe the output of the program whenever you press the interrupt button. Explain all the steps you would need to follow to place the interrupt service routine at address 2000h rather than 100h.

For additional credit: write your own interrupt service routine that does something simple. Run the main program and press the interrupt button to test your code. Verify that your ISR works properly.

3.6.6 Machine Language Programming & Instruction Encoding Exercises

To this point you have been creating machine language programs with SIMx86's built-in assembler. An assembler is a program that translates an ASCII source file containing textual representations of a program into the actual machine code. The assembler program saves you a considerable amount of work by translating human readable instructions into machine code. Although tedious, you can perform this translation yourself. In this exercise you will create some very short *machine language* programs by encoding the instructions and entering their hexadecimal opcodes into memory on the memory screen.

Using the instruction encodings found in Figure 3.19, Figure 3.20, Figure 3.21, and Figure 3.22, write the hexadecimal values for the opcodes beside each of the following instructions:



You can assume that the program starts at address zero and, therefore, label "a" will be at address 0003 since the mov cx, 0 instruction is three bytes long.

For your lab report: enter the hexadecimal opcodes and operands into memory starting at location zero using the Memory editor screen. Dump these values and include them in your lab report. Switch to the Emulator screen and disassemble the code starting at address zero. Verify that this code is the same as the assembly code above. Print a copy of the disassembled code and include it in your lab report. Run the program and verify that it works properly.

3.6.7 Self Modifying Code Exercises

In the previous laboratory exercise, you discovered that the system doesn't really differentiate data and instructions in memory. You were able to enter hexadecimal data and the x86 processor treats it as a sequence of executable instructions. It is also possible for a program to store data into memory and then execute it. A program is *self-modifying* if it creates or modifies some of the instructions it executes.

Consider the following x86 program (EX6.x86):

```
        sub     ax, ax
        mov     [100], ax

a:      mov     ax, [100]
        cmp     ax, 0
        je      b
        halt

b:      mov     ax, 00c6
        mov     [100], ax
        mov     ax, 0710
        mov     [102], ax
        mov     ax, a6a0
        mov     [104], ax
        mov     ax, 1000
        mov     [106], ax
        mov     ax, 8007
        mov     [108], ax
        mov     ax, 00e6
        mov     [10a], ax
        mov     ax, 0e10
        mov     [10c], ax
        mov     ax, 4
        mov     [10e], ax
        jmp     100
```

This program writes the following code to location 100 and then executes it:

```
        mov     ax, [1000]
        put
        add     ax, ax
        add     ax, [1000]
        put
        sub     ax, ax
        mov     [1000], ax
        jmp     0004                ;0004 is the address of the A: label.
```

For your lab report: execute the EX7.x86 program and verify that it generates the above code at location 100.

Although this program demonstrates the principle of self-modifying code, it hardly does anything useful. As a general rule, one would not use self-modifying code in the manner above, where one segment writes some sequence of instructions and then executes them. Instead, most programs that use self-modifying code only modify existing instructions and often only the operands of those instructions.

Self-modifying code is rarely found in modern assembly language programs. Programs that are self-modifying are hard to read and understand, difficult to debug, and often unstable. Programmers often resort to self-modifying code when the CPU's architecture lacks sufficient power to achieve a desired goal. The later Intel 80x86 processors do not lack for instructions or addressing modes, so it is very rare to find 80x86 programs that use self-modifying code[23]. The x86 processors, however, have a very weak instruction set, so there are actually a couple of instances where self-modifying code may prove useful.

A good example of an architectural deficiency where the x86 is lacking is with respect to subroutines. The x86 instruction set does not provide any (direct) way to call and return from a subroutine. However, you can easily simulate a call and return using the jmp instruction and self-modifying code. Consider the following x86 "subroutine" that sits at location 100h in memory:

```
; Integer to Binary converter.
; Expects an unsigned integer value in AX.
; Converts this to a string of zeros and ones storing this string of
; values into memory starting at location 1000h.

       mov     bx, 1000            ;Starting address of string.
       mov     cx, 10              ;16 (10h) digits in a word.
a:     mov     dx, 0               ;Assume current bit is zero.
       cmp     ax, 8000            ;See if AX's H.O. bit is zero or one.
       jb      b                   ;Branch if AX'x H.O. bit is zero.
       mov     dx, 1               ;AX's H.O. bit is one, set that here.
b:     mov     [bx], dx            ;Store zero or one to next string loc.
       add     bx, 1               ;Bump BX up to next string location.
       add     ax, ax              ;AX = AX *2 (shift left operation).
       sub     cx, 1               ;Count off 16 bits.
       cmp     cx, 0               ;Repeat 16 times.
       ja      a
       jmp     0                   ;Return to caller via self-mod code.
```

The only instruction that a program will modify in this subroutine is the very last jmp instruction. This jump instruction must transfer control to the first instruction beyond the jmp in the calling code that transfers control to this subroutine; that is, the caller must store the return address into the operand of the jmp instruction in the code above. As it turns out, the jmp instruction is at address 120h (assuming the code above starts at location 100h). Therefore, the caller must store the return address into location 121h (the operand of the jmp instruction). The following sample "main" program makes three calls to the "subroutine" above:

```
       mov     ax, 000c            ;Address of the BRK instr below.
       mov     [121], ax           ;Store into JMP as return address.
       mov     ax, 1234            ;Convert 1234h to binary.
       jmp     100                 ;"Call" the subroutine above.
       brk                         ;Pause to let the user examine 1000h.

       mov     ax, 0019            ;Address of the brk instr below.
       mov     [121], ax
       mov     ax, fdeb            ;Convert 0FDEBh to binary.
       jmp     100
       brk

       mov     ax, 26              ;Address of the halt instr below.
       mov     [121], ax
       mov     ax, 2345            ;Convert 2345h to binary.
       jmp     100

       halt
```

23. Many viruses and copy protection programs use self modifying code to make it difficult to detect or bypass them.

Load the subroutine (EX7s.x86) into SIMx86 and assemble it starting at location 100h. Next, load the main program (EX7m.x86) into memory and assemble it starting at location zero. Switch to the Emulator screen and verify that all the return addresses (0ch, 19h, and 26h) are correct. Also verify that the return address needs to be written to location 121h. Next, run the program. The program will execute a brk instruction after each of the first two calls. The brk instruction pauses the program. At this point you can switch to the memory screen at look at locations 1000-100F in memory. They should contain the pseudo-binary conversion of the value passed to the subroutine. Once you verify that the conversion is correct, switch back to the Emulator screen and press the Run button to continue program execution after the brk.

For your lab report: describe how self-modifying code works and explain in detail how this code uses self-modifying code to simulate call and return instructions. Explain the modifications you would need to make to move the main program to address 800h and the subroutine to location 900h.

For additional credit: Actually change the program and subroutine so that they work properly at the addresses above (800h and 900h).

3.7 Programming Projects

Note: You are to write these programs in x86 assembly language code using the SIMx86 program. Include a specification document, a test plan, a program listing, and sample output with your program submissions

1) The x86 instruction set does not include a multiply instruction. Write a short program that reads two values from the user and displays their product (hint: remember that multiplication is just repeated addition).

2) Create a callable subroutine that performs the multplication inproblem (1) above. Pass the two values to multiple to the subroutine in the ax and bx registers. Return the product in the cx register. Use the self-modifying code technique found in the section “Self Modifying Code Exercises” on page 136.

3) Write a program that reads two two-bit numbers from switches (FFF0/FFF2) and (FFF4/FFF6). Treating these bits as logical values, your code should compute the three-bit sum of these two values (two-bit result plus a carry). Use the logic equations for the full adder from the previous chapter. *Do not simply add these values using the x86 add instruction.* Display the three-bit result on LEDs FFF8, FFFA, and FFFC.

4) Write a subroutine that expects an address in BX, a count in CX, and a value in AX. It should write CX copies of AX to successive words in memory starting at address BX. Write a main program that calls this subroutine several times with different addresses. Use the self-modifying code subroutine call and return mechanism described in the laboratory exercises.

5) Write the generic logic function for the x86 processor (see Chapter Two). Hint: add ax, ax does a shift left on the value in ax. You can test to see if the high order bit is set by checking to see if ax is greater than 8000h.

6) Write a program that reads the generic function number for a four-input function from the user and then continually reads the switches and writes the result to an LED.

7) Write a program that scans an array of words starting at address 1000h and memory, of the length specified by the value in cx, and locates the maximum value in that array. Display the value after scanning the array.

8) Write a program that computes the two’s complement of an array of values starting at location 1000h. CX should contain the number of values in the array. Assume each array element is a two-byte integer.

9) Write a “light show” program that displays a “light show” on the SIMx86’s LEDs. It should accomplish this by writing a set of values to the LEDs, delaying for some time

period (by executing an empty loop) and then repeating the process over and over again. Store the values to write to the LEDs in an array in memory and fetch a new set of LED values from this array on each loop iteration.

10) Write a simple program that *sorts* the words in memory locations 1000..10FF in ascending order. You can use a simple *insertion sort* algorithm. The Pascal code for such a sort is

```
for i := 0 to n-1 do
       for j := i+1 to n do
                 if (memory[i] > memory[j]) then
                 begin
                          temp := memory[i];
                          memory[i] := memory[j];
                          memory[j] := temp;
                 end;
```

3.8 Summary

Writing good assembly language programs requires a strong knowledge of the underlying hardware. Simply knowing the instruction set is insufficient. To produce the best programs, you must understand how the hardware executes your programs and accesses data.

Most modern computer systems store programs and data in the same memory space (the *Von Neumann architecture*). Like most Von Neumann machines, a typical 80x86 system has three major components: the *central processing unit* (CPU), *input/output* (I/O), and *memory*. See:

- "The Basic System Components" on page 83

Data travels between the CPU, I/O devices, and memory on the *system bus*. There are three major busses employed by the 80x86 family, the *address bus*, the *data bus*, and the *control bus*. The address bus carries a binary number which specifies which memory location or I/O port the CPU wishes to access; the data bus carries data between the CPU and memory or I/O; the control bus carries important signals which determine whether the CPU is reading or writing memory data or accessing an I/O port. See:

- "The System Bus" on page 84
- "The Data Bus" on page 84
- "The Address Bus" on page 86
- "The Control Bus" on page 86

The number of data lines on the data bus determines the *size* of a processor. When we say that a processor is an *eight bit processor* we mean that it has eight data lines on its data bus. The size of the data which the processor can handle internally on the CPU does not affect the size of that CPU. See:

- "The Data Bus" on page 84
- "The "Size" of a Processor" on page 85

The address bus transmits a binary number from the CPU to memory and I/O to select a particular memory element or I/O port. The number of lines on the address bus sets the maximum number of memory locations the CPU can access. Typical address bus sizes on the 80x86 CPUs are 20, 24, and 32 bits. See:

- "The Address Bus" on page 86

The 80x86 CPUs also have a control bus which contains various signals necessary for the proper operation of the system. The system clock, read/write control signals, and I/O or memory control signals are some samples of the many lines which appear on the control bus. See:

- "The Control Bus" on page 86

The memory subsystem is where the CPU stores program instructions and data. On 80x86 based systems, memory appears as a linear array of bytes, each with its own unique address. The address of the first byte in memory is zero, and the address of the last available byte in memory is 2^n-1, where *n* is the number of lines on the address bus. The 80x86 stores words in two consecutive memory locations. The L.O. byte of the word is at the lowest address of those two bytes; the H.O. byte immediately follows the first at the next highest address. Although a word consumes two memory addresses, when dealing with words we simply use the address of its L.O. byte as the address of that word. Double words consume four consecutive bytes in memory. The L.O. byte appears at the lowest address of the four, the H.O. byte appears at the highest address. The "address" of the double word is the address of its L.O. byte. See:

- "The Memory Subsystem" on page 87

CPUs with 16, 32, or 64 bit data busses generally organize memory in *banks*. A 16 bit memory subsystem uses two banks of eight bits each, a 32 bit memory subsystem uses four banks of eight bits each, and a 64 bit system uses eight banks of eight bits each. Accessing a word or double word at the same address within all the banks is faster than accessing an object which is split across two addresses in the different banks. Therefore, you should attempt to align word data so that it begins on an even address and double word data so that it begins on an address which is evenly divisible by four. You may place byte data at any address. See:

- "The Memory Subsystem" on page 87

The 80x86 CPUs provide a separate 16 bit I/O address space which lets the CPU access any one of 65,536 different I/O ports. A typical I/O device connected to the IBM PC only uses 10 of these address lines, limiting the system to 1,024 different I/O ports. The major benefit to using an I/O address space rather than mapping all I/O devices to memory space is that the I/O devices need not infringe upon the addressable memory space. To differentiate I/O and memory accesses, there are special control lines on the system bus. See:

- "The Control Bus" on page 86
- "The I/O Subsystem" on page 92

The system clock controls the speed at which the processor performs basic operations. Most CPU activities occur on the rising or falling edge of this clock. Examples include instruction execution, memory access, and checking for wait states. The faster the system clock runs, the faster your program will execute; however, your memory must be as fast as the system clock or you will need to introduce wait states, which slow the system back down. See:

- "System Timing" on page 92
- "The System Clock" on page 92
- "Memory Access and the System Clock" on page 93
- "Wait States" on page 95

Most programs exhibit a *locality of reference*. They either access the same memory location repeatedly within a small period of time (*temporal locality*) or they access neighboring memory locations during a short time period (*spatial locality*). A *cache memory subsystem* exploits this phenomenon to reduce wait states in a system. A small cache memory system can achieve an 80-95% hit ratio. *Two-level caching systems* use two different caches (typically one on the CPU chip and one off chip) to achieve even better system performance. See:

- "Cache Memory" on page 96

CPUs, such as those in the 80x86 family, break the execution of a machine instruction down into several distinct steps, each requiring one clock cycle. These steps include fetching an instruction opcode, decoding that opcode, fetching operands for the instruction, computing memory addresses, accessing memory, performing the basic operation, and storing the result away. On a very simplistic CPU, a simple instruction may take several clock cycles. The best way to improve the performance of a CPU is to execute several

internal operations in parallel with one another. A simple scheme is to put an instruction prefetch queue on the CPU. This allows you to overlap opcode fetching and decoding with instruction execution, often cutting the execution time in half. Another alternative is to use an instruction pipeline so you can execute several instructions in parallel. Finally, you can design a superscalar CPU which executes two or more instructions concurrently. These techniques will all improve the running time of your programs. See:

- "The 886 Processor" on page 110
- "The 8286 Processor" on page 110
- "The 8486 Processor" on page 116
- "The 8686 Processor" on page 123

Although pipelined and superscalar CPUs improve overall system performance, extracting the best performance from such complex CPUs requires careful planning by the programmer. Pipeline stalls and hazards can cause a major loss of performance in poorly organized programs. By carefully organizing the sequence of the instructions in your programs you can make your programs run as much as two to three times faster. See:

- "The 8486 Pipeline" on page 117
- "Stalls in a Pipeline" on page 118
- "Cache, the Prefetch Queue, and the 8486" on page 119
- "Hazards on the 8486" on page 122
- "The 8686 Processor" on page 123

The I/O subsystem is the third major component of a Von Neumann machine (memory and the CPU being the other two). There are three primary ways to move data between the computer system and the outside world: I/O-mapped input/output, memory-mapped input/output, and direct memory access (DMA). For more information, see:

- "I/O (Input/Output)" on page 124

To improve system performance, most modern computer systems use interrupts to alert the CPU when an I/O operation is complete. This allows the CPU to continue with other processing rather than waiting for an I/O operation to complete (polling the I/O port). For more information on interrupts and polled I/O operatoins, see:

- "Interrupts and Polled I/O" on page 126

3.9 Questions

1. What three components make up Von Neumann Machines?
2. What is the purpose of

   a) The system bus
   b) The address bus
   c) The data bus
   d) The control bus

3. Which bus defines the “size” of the processor?
4. Which bus controls how much memory you can have?
5. Does the size of the data bus control the maximum value the CPU can process? Explain.
6. What are the data bus sizes of:

   a) 8088 b) 8086 c) 80286 d) 80386sx
   e) 80386 f) 80486 g) 80586 / Pentium

7. What are the address bus sizes of the above processors?
8. How many “banks” of memory do each of the above processors have?
9. Explain how to store a word in byte addressable memory (that is, at what addresses). Explain how to store a double word.
10. How many memory operations will it take to read a word from the following addresses on the following processors?

Table 21: Memory Cycles for Word Accesses

	100	101	102	103	104	105
8088						
80286						
80386						

11. Repeat the above for double words

Table 22: Memory Cycles for Doubleword Accesses

	100	101	102	103	104	105
8088						
80286						
80386						

12. Explain which addresses are best for byte, word, and doubleword variables on an 8088, 80286, and 80386 processor.
13. How many different I / O locations can you address on the 80x86 chip? How many are typically available on a PC?
14. What is the purpose of the system clock?

15. What is a clock cycle?

16. What is the relationship between clock frequency and the clock period?

17. How many clock cycles are required for each of the following to read a byte from memory?

    a) 8088 b) 8086 c) 80486

18. What does the term “memory access time” mean?

19. What is a *wait state?*

20. If you are running an 80486 at the following clock speeds, how many wait states are required if you are using 80ns RAM (assuming no other delays)?

    a) 20 MHz b) 25 MHz c) 33 MHz d) 50 MHz e) 100 MHz

21. If your CPU runs at 50 MHz, 20ns RAM probably won’t be fast enough to operate at zero wait states. Explain why.

22. Since sub-10ns RAM is available, why aren’t all system zero wait state systems?

23. Explain how the cache operates to save some wait states.

24. What is the difference between spatial and temporal locality of reference?

25. Explain where temporal and spatial locality of reference occur in the following Pascal code:

```
while i < 10 do begin
                x := x * i;
                i := i + 1;
end;
```

26. How does cache memory improve the performance of a section of code exhibiting spatial locality of reference?

27. Under what circumstances is a cache not going to save you any wait states?

28. What is the effective (average) number of wait states the following systems will operate under?

    a) 80% cache hit ratio, 10 wait states (WS) for memory, 0 WS for cache.

    b) 90% cache hit ratio; 7 WS for memory; 0 WS for cache.

    c) 95% cache hit ratio; 10 WS memory; 1 WS cache.

    d) 50% cache hit ratio; 2 WS memory; 0 WS cache.

29. What is the purpose of a two level caching system? What does it save?

30. What is the effective number of wait states for the following systems?

    a) 80% primary cache hit ratio (HR) zero WS; 95% secondary cache HR with 2 WS; 10 WS for main memory access.

    b) 50% primary cache HR, zero WS; 98% secondary cache HR, one WS; five WS for main memory access.

    c) 95% primary cache HR, one WS; 98% secondary cache HR, 4 WS; 10 WS for main memory access.

31. Explain the purpose of the *bus interface unit*, the *execution unit*, and the *control unit.*

32. Why does it take more than one clock cycle to execute an instruction. Give some x86 examples.

33. How does a prefetch queue save you time? Give some examples.

34. How does a pipeline allow you to (seemingly) execute one instruction per clock cycle? Give an example.
35. What is a hazard?
36. What happens on the 8486 when a hazard occurs?
37. How can you eliminate the effects of a hazard?
38. How does a jump (JMP/Jcc) instruction affect

    a) The prefetch queue.
    b) The pipeline.

39. What is a pipeline stall?
40. Besides the obvious benefit of reducing wait states, how can a cache improve the performance of a pipelined system?
41. What is a Harvard Architecture Machine?
42. What does a superscalar CPU do to speed up execution?
43. What are the two main techniques you should use on a superscalar CPU to ensure your code runs as quickly as possible? (note: these are mechanical details, “Better Algorithms” doesn’t count here).
44. What is an interrupt? How does it improved system performance?
45. What is polled I/O?
46. What is the difference between memory-mapped and I/O mapped I/O?
47. DMA is a special case of memory-mapped I/O. Explain.

Memory Layout and Access Chapter Four

Chapter One discussed the basic format for data in memory. Chapter Three covered how a computer system physically organizes that data. This chapter discusses how the 80x86 CPUs access data in memory.

4.0 Chapter Overview

This chapter forms an important bridge between sections one and two (Machine Organization and Basic Assembly Language, respectively). From the point of view of machine organization, this chapter discusses memory addressing, memory organization, CPU addressing modes, and data representation in memory. From the assembly language programming point of view, this chapter discusses the 80x86 register sets, the 80x86 memory addressing modes, and composite data types. This is a pivotal chapter. If you do not understand the material in this chapter, you will have difficulty understanding the chapters that follow. Therefore, you should study this chapter carefully before proceeding.

This chapter begins by discussing the registers on the 80x86 processors. These processors provide a set of general purpose registers, segment registers, and some special purpose registers. Certain members of the family provide additional registers, although typical application do not use them.

After presenting the registers, this chapter describes memory organization and segmentation on the 80x86. Segmentation is a difficult concept to many beginning 80x86 assembly language programmers. Indeed, this text tends to avoid using segmented addressing throughout the introductory chapters. Nevertheless, segmentation is a powerful concept that you must become comfortable with if you intend to write non-trivial 80x86 programs.

80x86 memory addressing modes are, perhaps, the most important topic in this chapter. Unless you completely master the use of these addressing modes, you will not be able to write reasonable assembly language programs. Do not progress beyond this section of the text until you are comfortable with the 8086 addressing modes. This chapter also discusses the 80386 (and later) extended addressing modes. Knowing these addressing modes is not that important for now, but if you do learn them you can use them to save some time when writing code for 80386 and later processors.

This chapter also introduces a handful of 80x86 instructions. Although the five or so instructions this chapter uses are insufficient for writing real assembly language programs, they do provide a sufficient set of instructions to let you manipulate variables and data structures – the subject of the next chapter.

4.1 The 80x86 CPUs:A Programmer's View

Now it's time to discuss some real processors: the 8088 / 8086, 80188 / 80186, 80286, and 80386 / 80486 / 80586 / Pentium. Chapter Three dealt with many hardware aspects of a computer system. While these hardware components affect the way you should write software, there is more to a CPU than bus cycles and pipelines. It's time to look at those components of the CPU which are most visible to you, the assembly language programmer.

The most visible component of the CPU is the register set. Like our hypothetical processors, the 80x86 chips have a set of on-board registers. The register set for each processor in the 80x86 family is a superset of those in the preceding CPUs. The best place to start is with the register set for the 8088, 8086, 80188, and 80186 since these four processors have the same registers. In the discussion which follows, the term "8086" will imply any of these four CPUs.

Intel's designers have classified the registers on the 8086 into three categories: general purpose registers, segment registers, and miscellaneous registers. The general purpose registers are those which may appear as operands of the arithmetic, logical, and related instructions. Although these registers are "general purpose", every one has its own special purpose. Intel uses the term "general purpose" loosely. The 8086 uses the segment registers to access blocks of memory called, surprisingly enough, segments. See "Segments on the 80x86" on page 151 for more details on the exact nature of the segment registers. The final class of 8086 registers are the miscellaneous registers. There are two special registers in this group which we'll discuss shortly.

4.1.1 8086 General Purpose Registers

There are eight 16 bit general purpose registers on the 8086: ax, bx, cx, dx, si, di, bp, and sp. While you can use many of these registers interchangeably in a computation, many instructions work more efficiently or absolutely require a specific register from this group. So much for general purpose.

The ax register (*Accumulator*) is where most arithmetic and logical computations take place. Although you can do most arithmetic and logical operations in other registers, it is often more efficient to use the ax register for such computations. The bx register (*Base*) has some special purposes as well. It is commonly used to hold indirect addresses, much like the bx register on the x86 processors. The cx register (*Count*), as its name implies, counts things. You often use it to count off the number of iterations in a loop or specify the number of characters in a string. The dx register (*Data*) has two special purposes: it holds the overflow from certain arithmetic operations, and it holds I/O addresses when accessing data on the 80x86 I/O bus.

The si and di registers (*Source Index* and *Destination Index*) have some special purposes as well. You may use these registers as pointers (much like the bx register) to indirectly access memory. You'll also use these registers with the 8086 string instructions when processing character strings.

The bp register (*Base Pointer*) is similar to the bx register. You'll generally use this register to access parameters and local variables in a procedure.

The sp register (*Stack Pointer*) has a very special purpose – it maintains the *program stack*. Normally, you would not use this register for arithmetic computations. The proper operation of most programs depends upon the careful use of this register.

Besides the eight 16 bit registers, the 8086 CPUs also have eight 8 bit registers. Intel calls these registers al, ah, bl, bh, cl, ch, dl, and dh. You've probably noticed a similarity between these names and the names of some 16 bit registers (ax, bx, cx, and dx, to be exact). The eight bit registers are not independent. al stands for "ax's L.O. byte." ah stands for "ax's H.O. byte." The names of the other eight bit registers mean the same thing with respect to bx, cx, and dx. Figure 4.1 shows the general purpose register set.

Note that the eight bit registers do not form an independent register set. Modifying al will change the value of ax; so will modifying ah. The value of al exactly corresponds to bits zero through seven of ax. The value of ah corresponds to bits eight through fifteen of ax. Therefore any modification to al or ah will modify the value of ax. Likewise, modifying ax will change *both* al and ah. Note, however, that changing al will not affect the value of ah, and vice versa. This statement applies to bx/bl/bh, cx/cl/ch, and dx/dl/dh as well.

The si, di, bp, and sp registers are only 16 bits. There is no way to directly access the individual bytes of these registers as you can the low and high order bytes of ax, bx, cx, and dx.

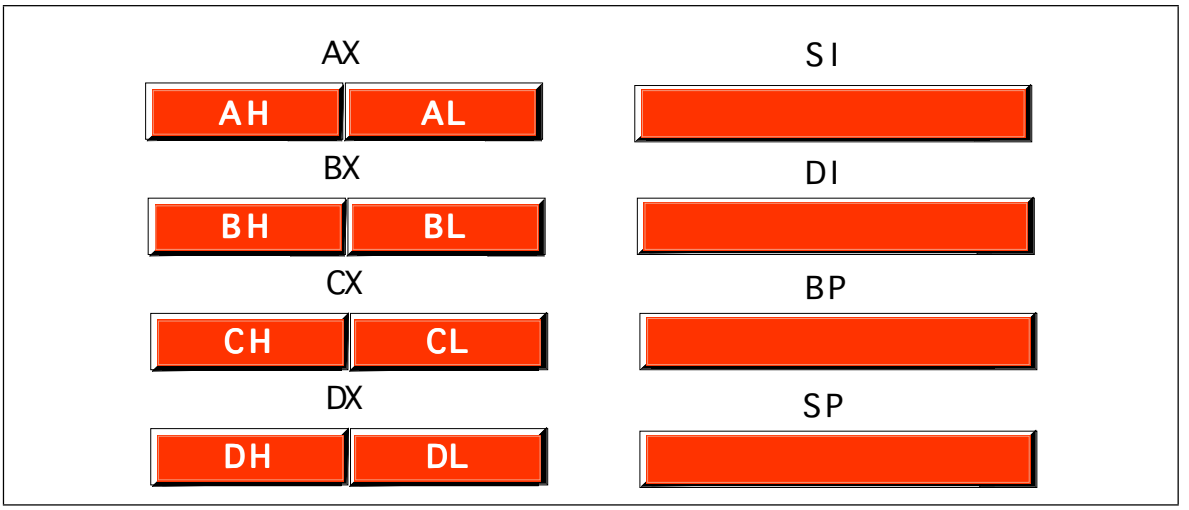


Figure 4.1 8086 Register Set

4.1.2 8086 Segment Registers

The 8086 has four special *segment registers*: cs, ds, es, and ss. These stand for *Code Segment*, *Data Segment*, *Extra Segment*, and *Stack Segment*, respectively. These registers are all 16 bits wide. They deal with selecting blocks (segments) of main memory. A segment register (e.g., cs) points at the beginning of a segment in memory.

Segments of memory on the 8086 can be no larger than 65,536 bytes long. This infamous "64K segment limitation" has disturbed many a programmer. We'll see some problems with this 64K limitation, and some solutions to those problems, later.

The cs register points at the segment containing the currently executing machine instructions. Note that, despite the 64K segment limitation, 8086 programs can be longer than 64K. You simply need multiple code segments in memory. Since you can change the value of the cs register, you can switch to a new code segment when you want to execute the code located there.

The data segment register, ds, generally points at global variables for the program. Again, you're limited to 65,536 bytes of data in the data segment; but you can always change the value of the ds register to access additional data in other segments.

The extra segment register, es, is exactly that – an extra segment register. 8086 programs often use this segment register to gain access to segments when it is difficult or impossible to modify the other segment registers.

The ss register points at the segment containing the 8086 *stack*. The stack is where the 8086 stores important machine state information, subroutine return addresses, procedure parameters, and local variables. In general, you do not modify the stack segment register because too many things in the system depend upon it.

Although it is theoretically possible to store data in the segment registers, this is never a good idea. The segment registers have a very special purpose – pointing at accessible blocks of memory. Any attempt to use the registers for any other purpose may result in considerable grief, especially if you intend to move up to a better CPU like the 80386.

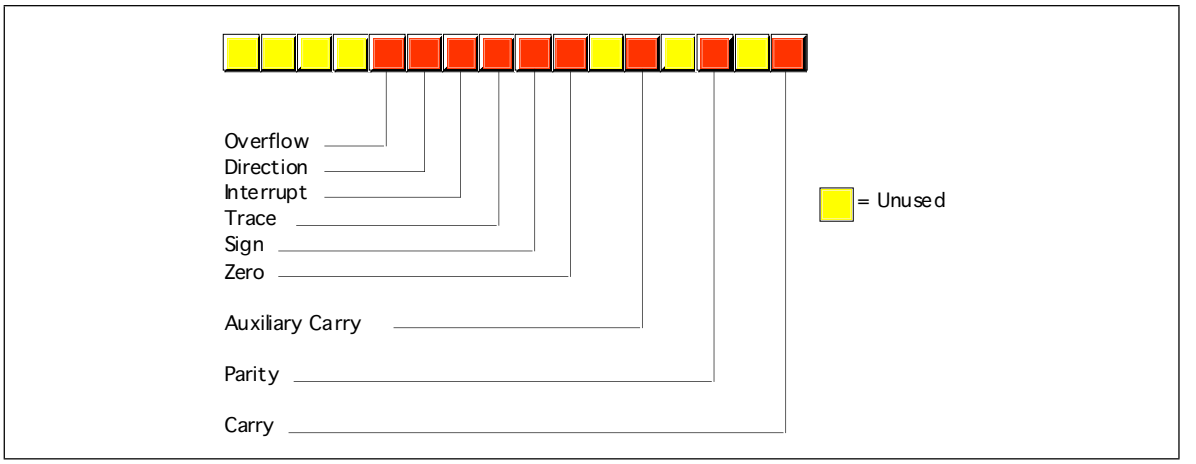


Figure 4.2 8086 Flags Register

4.1.3 8086 Special Purpose Registers

There are two special purpose registers on the 8086 CPU: the instruction pointer (ip) and the flags register. You do not access these registers the same way you access the other 8086 registers. Instead, the CPU generally manipulates these registers directly.

The ip register is the equivalent of the ip register on the x86 processors – it contains the address of the currently executing instruction. This is a 16 bit register which provides a pointer into the current code segment (16 bits lets you select any one of 65,536 different memory locations). We'll come back to this register when we discuss the control transfer instructions later.

The flags register is unlike the other registers on the 8086. The other registers hold eight or 16 bit values. The flags register is simply an eclectic collection of one bit values which help determine the current state of the processor. Although the flags register is 16 bits wide, the 8086 uses only nine of those bits. Of these flags, four flags you use all the time: zero, carry, sign, and overflow. These flags are the 8086 *condition codes*. The flags register appears in Figure 4.2.

4.1.4 80286 Registers

The 80286 microprocessor adds one major programmer-visible feature to the 8086 – protected mode operation. This text will not cover the 80286 protected mode of operation for a variety of reasons. First, the protected mode of the 80286 was poorly designed. Second, it is of interest only to programmers who are writing their own operating system or low-level systems programs for such operating systems. Even if you are writing software for a protected mode operating system like UNIX or OS/2, you would not use the protected mode features of the 80286. Nonetheless, it's worthwhile to point out the extra registers and status flags present on the 80286 just in case you come across them.

There are three additional bits present in the 80286 flags register. The I/O Privilege Level is a two bit value (bits 12 and 13). It specifies one of four different privilege levels necessary to perform I/O operations. These two bits generally contain 00b when operating in *real mode* on the 80286 (the 8086 emulation mode). The NT (*nested task*) flag controls the operation of an interrupt return (IRET) instruction. NT is normally zero for real-mode programs.

Besides the extra bits in the flags register, the 80286 also has five additional registers used by an operating system to support memory management and multiple processes: the

machine status word (msw), the global descriptor table register (gdtr), the local descriptor table register (ldtr), the interrupt descriptor table register (idtr) and the task register (tr).

About the only use a typical application program has for the protected mode on the 80286 is to access more than one megabyte of RAM. However, as the 80286 is now virtually obsolete, and there are better ways to access more memory on later processors, programmers rarely use this form of protected mode.

4.1.5 80386/80486 Registers

The 80386 processor dramatically extended the 8086 register set. In addition to all the registers on the 80286 (and therefore, the 8086), the 80386 added several new registers and extended the definition of the existing registers. The 80486 did not add any new registers to the 80386's basic register set, but it did define a few bits in some registers left undefined by the 80386.

The most important change, from the programmer's point of view, to the 80386 was the introduction of a 32 bit register set. The ax, bx, cx, dx, si, di, bp, sp, flags, and ip registers were all extended to 32 bits. The 80386 calls these new 32 bit versions eax, ebx, ecx, edx, esi, edi, ebp, esp, eflags, and eip to differentiate them from their 16 bit versions (which are still available on the 80386). Besides the 32 bit registers, the 80386 also provides two new 16 bit segment registers, fs and gs, which allow the programmer to concurrently access six different segments in memory without reloading a segment register. Note that all the segment registers on the 80386 are 16 bits. The 80386 did not extend the segment registers to 32 bits as it did the other registers.

The 80386 did not make any changes to the bits in the flags register. Instead, it extended the flags register to 32 bits (the "eflags" register) and defined bits 16 and 17. Bit 16 is the debug resume flag (RF) used with the set of 80386 debug registers. Bit 17 is the Virtual 8086 mode flag (VM) which determines whether the processor is operating in virtual-86 mode (which simulates an 8086) or standard protected mode. The 80486 adds a third bit to the eflags register at position 18 – the alignment check flag. Along with control register zero (CR0) on the 80486, this flag forces a trap (program abort) whenever the processor accesses non-aligned data (e.g., a word on an odd address or a double word at an address which is not an even multiple of four).

The 80386 added four control registers: CR0-CR3. These registers extend the msw register of the 80286 (the 80386 emulates the 80286 msw register for compatibility, but the information really appears in the CRx registers). On the 80386 and 80486 these registers control functions such as paged memory management, cache enable/disable/operation (80486 only), protected mode operation, and more.

The 80386/486 also adds eight *debugging* registers. A debugging program like Microsoft Codeview or the Turbo Debugger can use these registers to set breakpoints when you are trying to locate errors within a program. While you would not use these registers in an application program, you'll often find that using such a debugger reduces the time it takes to eradicate bugs from your programs. Of course, a debugger which accesses these registers will only function properly on an 80386 or later processor.

Finally, the 80386/486 processors add a set of test registers to the system which test the proper operation of the processor when the system powers up. Most likely, Intel put these registers on the chip to allow testing immediately after manufacture, but system designers can take advantage of these registers to do a power-on test.

For the most part, assembly language programmers need not concern themselves with the extra registers added to the 80386/486/Pentium processors. However, the 32 bit extensions and the extra segment registers are quite useful. To the application programmer, the *programming model* for the 80386/486/Pentium looks like that shown in Figure 4.3

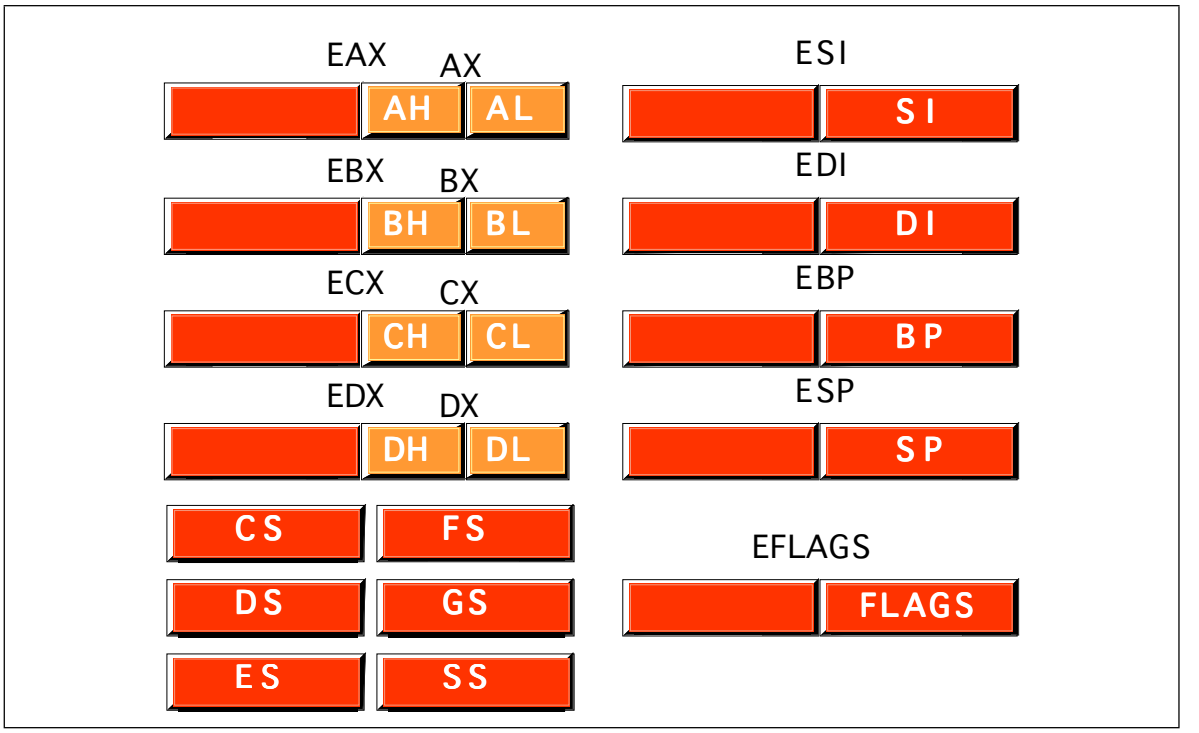


Figure 4.3 80386 Registers (Application Programmer Visible)

4.2 80x86 Physical Memory Organization

Chapter Three discussed the basic organization of a Von Neumann Architecture (VNA) computer system. In a typical VNA machine, the CPU connects to memory via the bus. The 80x86 selects some particular memory element using a binary number on the address bus. Another way to view memory is as an array of bytes. A Pascal data structure that roughly corresponds to memory would be:

```
Memory : array [0..MaxRAM] of byte;
```

The value on the address bus corresponds to the index supplied to this array. E.g., writing data to memory is equivalent to

```
Memory [address] := Value_to_Write;
```

Reading data from memory is equivalent to

```
Value_Read := Memory [address];
```

Different 80x86 CPUs have different address busses that control the maximum number of elements in the memory array (see “The Address Bus” on page 86). However, regardless of the number of address lines on the bus, most computer systems do *not* have one byte of memory for each addressable location. For example, 80386 processors have 32 address lines allowing up to four gigabytes of memory. Very few 80386 systems actually have four gigabytes. Usually, you’ll find one to 256 megabytes in an 80x86 based system.

The first megabyte of memory, from address zero to 0FFFFFh is special on the 80x86. This corresponds to the entire address space of the 8088, 8086, 80186, and 80188 microprocessors. Most DOS programs limit their program and data addresses to locations in this range. Addresses limited to this range are named *real addresses* after the 80x86 *real mode*.

4.3 Segments on the 80x86

You cannot discuss memory addressing on the 80x86 processor family without first discussing segmentation. Among other things, segmentation provides a powerful memory management mechanism. It allows programmers to partition their programs into modules that operate independently of one another. Segments provide a way to easily implement object-oriented programs. Segments allow two processes to easily share data. All in all, segmentation is a really neat feature. On the other hand, if you ask ten programmers what they think of segmentation, at least nine of the ten will claim it's terrible. Why such a response?

Well, it turns out that segmentation provides one other nifty feature: it allows you to extend the addressability of a processor. In the case of the 8086, segmentation let Intel's designers extend the maximum addressable memory from 64K to one megabyte. Gee, that sounds good. Why is everyone complaining? Well, a little history lesson is in order to understand what went wrong.

In 1976, when Intel began designing the 8086 processor, memory was very expensive. Personal computers, such that they were at the time, typically had four thousand bytes of memory. Even when IBM introduced the PC five years later, 64K was still quite a bit of memory, one megabyte was a tremendous amount. Intel's designers felt that 64K memory would remain a large amount throughout the lifetime of the 8086. The only mistake they made was completely underestimating the lifetime of the 8086. They figured it would last about five years, like their earlier 8080 processor. They had plans for lots of other processors at the time, and "86" was not a suffix on the names of any of those. Intel figured they were set. Surely one megabyte would be more than enough to last until they came out with something better[1].

Unfortunately, Intel didn't count on the IBM PC and the massive amount of software to appear for it. By 1983, it was very clear that Intel could not abandon the 80x86 architecture. They were stuck with it, but by then people were running up against the one megabyte limit of 8086. So Intel gave us the 80286. This processor could address up to 16 megabytes of memory. Surely more than enough. The only problem was that all that wonderful software written for the IBM PC was written in such a way that it couldn't take advantage of any memory beyond one megabyte.

It turns out that the maximum amount of addressable memory is not everyone's main complaint. The real problem is that the 8086 was a 16 bit processor, with 16 bit registers and 16 bit addresses. This limited the processor to addressing 64K chunks of memory. Intel's clever use of segmentation extended this to one megabyte, but addressing more than 64K at one time takes some effort. Addressing more than 256K at one time takes a *lot* of effort.

Despite what you might have heard, segmentation is not bad. In fact, it is a really great memory management scheme. What is bad is Intel's 1976 implementation of segmentation still in use today. You can't blame Intel for this – they fixed the problem in the 80's with the release of the 80386. The real culprit is MS-DOS that forces programmers to continue to use 1976 style segmentation. Fortunately, newer operating systems such as Linux, UNIX, Windows 9x, Windows NT, and OS/2 don't suffer from the same problems as MS-DOS. Furthermore, users finally seem to be more willing to switch to these newer operating systems so programmers can take advantage of the new features of the 80x86 family.

With the history lesson aside, it's probably a good idea to figure out what segmentation is all about. Consider the current view of memory: it looks like a linear array of bytes. A single index (address) selects some particular byte from that array. Let's call this type of addressing *linear* or *flat* addressing. Segmented addressing uses two components to specify a memory location: a segment value and an offset within that segment. Ideally, the segment and offset values are independent of one another. The best way to describe

1. At the time, the iapx432 processor was their next big product. It died a slow and horrible death.

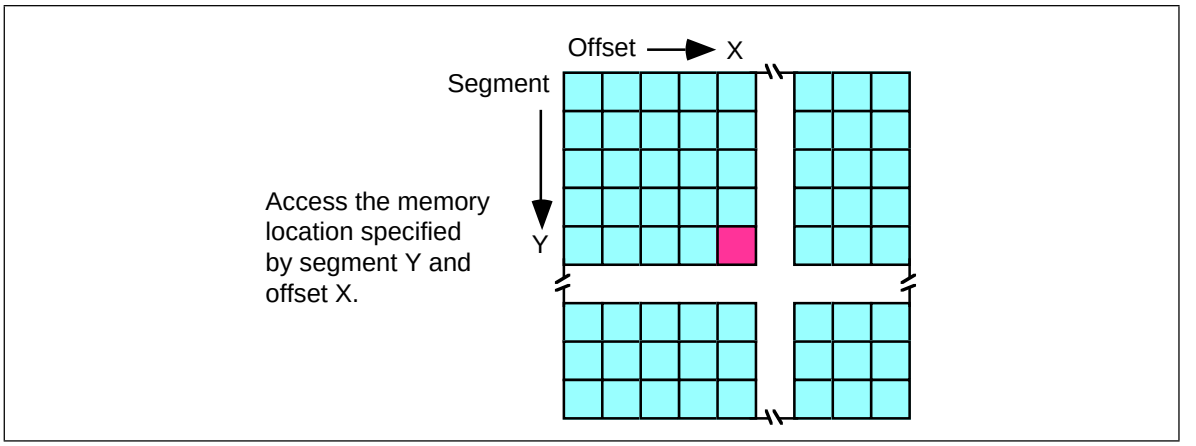


Figure 4.4 Segmented Addressing as a Two-Dimensional Process

segmented addressing is with a two-dimensional array. The segment provides one of the indices into the array, the offset provides the other (see Figure 4.4).

Now you may be wondering, “Why make this process more complex?” Linear addresses seem to work fine, why bother with this two dimensional addressing scheme? Well, let’s consider the way you typically write a program. If you were to write, say, a SIN(X) routine and you needed some temporary variables, you probably would not use global variables. Instead, you would use local variables inside the SIN(X) function. In a broad sense, this is one of the features that segmentation offers – the ability to attach blocks of variables (a segment) to a particular piece of code. You could, for example, have a segment containing local variables for SIN, a segment for SQRT, a segment for DRAW-Window, etc. Since the variables for SIN appear in the segment for SIN, it’s less likely your SIN routine will affect the variables belonging to the SQRT routine. Indeed, on the 80286 and later operating in *protected mode*, the CPU can *prevent* one routine from accidentally modifying the variables in a different segment.

A full segmented address contains a segment component and an offset component. This text will write segmented addresses as *segment*:*offset*. On the 8086 through the 80286, these two values are 16 bit constants. On the 80386 and later, the offset can be a 16 bit constant or a 32 bit constant.

The size of the offset limits the maximum size of a segment. On the 8086 with 16 bit offsets, a segment may be no longer than 64K; it could be smaller (and most segments are), but never larger. The 80386 and later processors allow 32 bit offsets with segments as large as four gigabytes.

The segment portion is 16 bits on all 80x86 processors. This lets a single program have up to 65,536 different segments in the program. Most programs have less than 16 segments (or thereabouts) so this isn’t a practical limitation.

Of course, despite the fact that the 80x86 family uses segmented addressing, the actual (*physical*) memory connected to the CPU is still a linear array of bytes. There is a function that converts the segment value to a physical memory address. The processor then adds the offset to this physical address to obtain the actual address of the data in memory. This text will refer to addresses in your programs as *segmented addresses* or *logical addresses*. The actual linear address that appears on the address bus is the *physical address* (see Figure 4.4).

On the 8086, 8088, 80186, and 80188 (and other processors operating in *real* mode), the function that maps a segment to a physical address is very simple. The CPU multiplies the segment value by sixteen (10h) and adds the offset portion. For example, consider the segmented address[2]: 1000:1F00. To convert this to a physical address you multiply the seg-

2. All segmented addresses in this text use the hexadecimal radix. Since this text will always use the hex radix for addresses, there is no need to append an “h” to the end of such values.

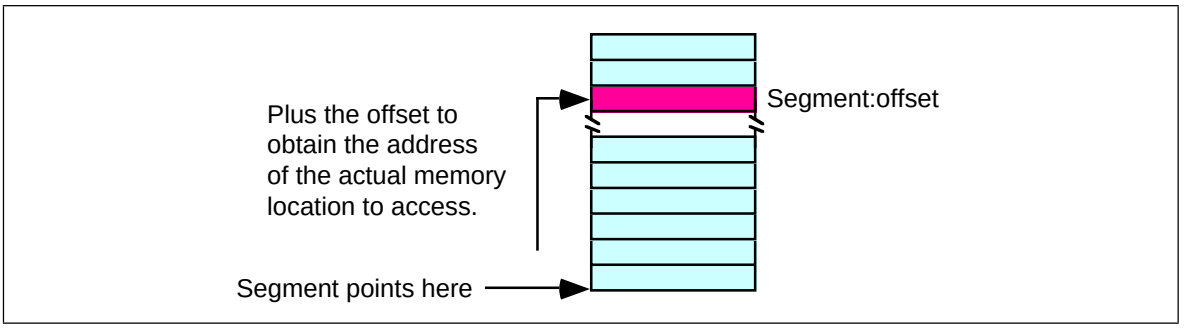


Figure 4.5 Segmented Addressing in Physical Memory

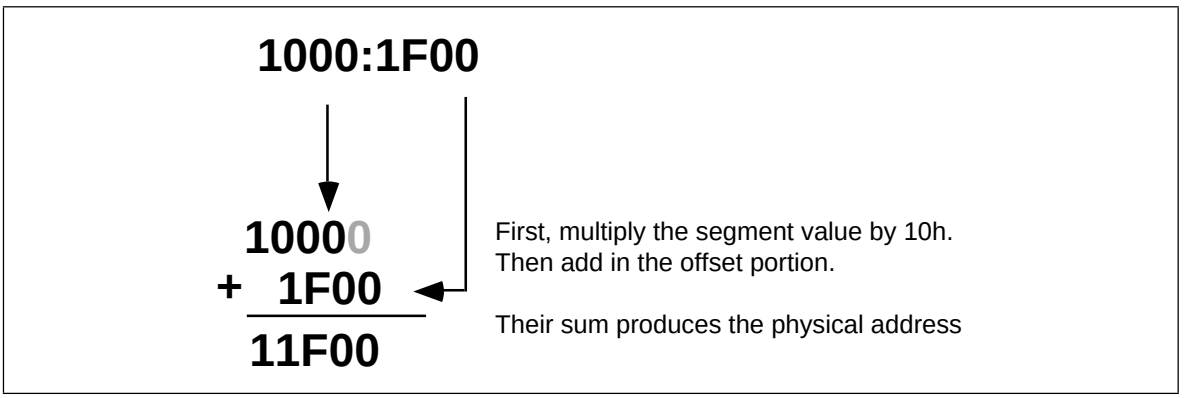


Figure 4.6 Converting a Logical Address to a Physical Address

ment value (1000h) by sixteen. Multiplying by the radix is very easy. Just append a zero to the end of the number. Appending a zero to 1000h produces 10000h. Add 1F00h to this to obtain 11F00h. So 11F00h is the physical address that corresponds to the segmented address 1000:1F00 (see Figure 4.4).

Warning: A very common mistake people make when performing this computation is to forget they are working in hexadecimal, not decimal. It is surprising to see how many people add 9+1 and get 10h rather than the correct answer 0Ah.

Intel, when designing the 80286 and later processors, did not extend the addressing by adding more bits to the segment registers. Instead, they changed the function the CPU uses to convert a logical address to a physical address. If you write code that depends on the "multiply by sixteen and add in the offset" function, your program will only work on an 80x86 processor operating in real mode, and you will be limited to one megabyte of memory[3].

In the 80286 and later processors, Intel introduced *protected mode segments*. Among other changes, Intel completely revamped the algorithm for mapping segments to the linear address space. Rather than using a function (such as multiplying the segment value by 10h), the protected mode processors use a *look up table* to compute the physical address. In protected mode, the 80286 and later processors use the segment value as the index into an array. The contents of the selected array element provide (among other things) the starting address for the segment. The CPU adds this value to the offset to obtain the physical address (see Figure 4.4).

Note that your applications cannot directly modify the segment descriptor table (the lookup table). The protected mode operating system (UNIX, Linux, Windows, OS/2, etc.) handles that operation.

3. Actually, you can also operate in V86 (virtual 86) mode on the 80386 and later, but you will still be limited to one megabyte addressable memory.

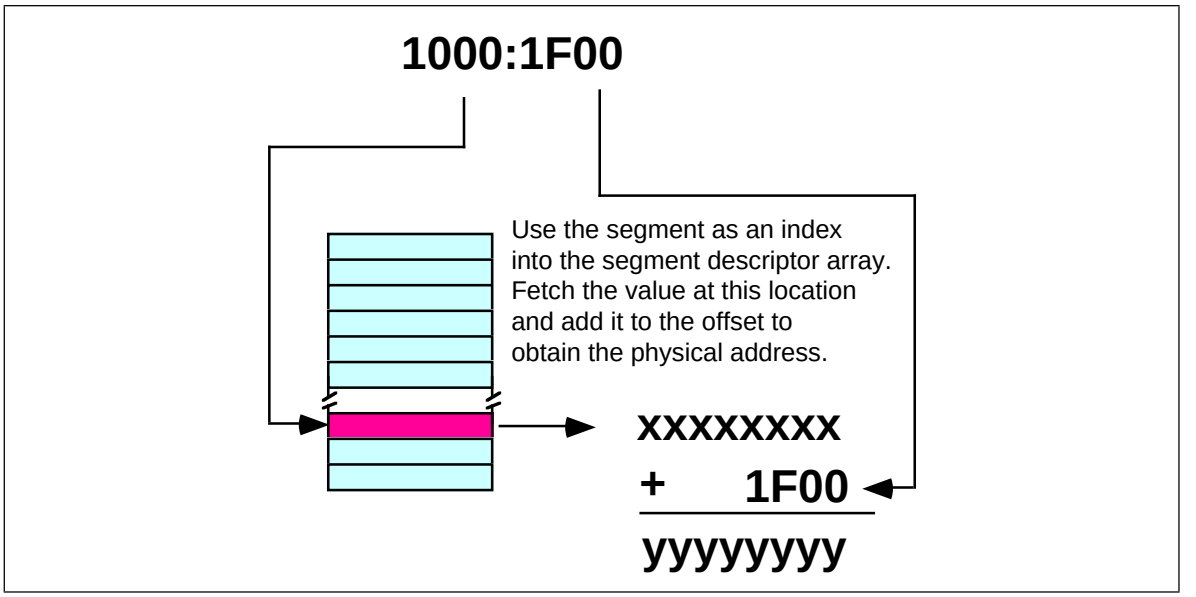


Figure 4.7 Converting a Logical Address to a Physical Address in Protected Mode

The best programs never assume that a segment is located at a particular spot in memory. You should leave it up to the operating system to place your programs into memory and not generate any segment addresses on your own.

4.4 Normalized Addresses on the 80x86

When operating in real mode, an interesting problem develops. You may refer to a single object in memory using several *different* addresses. Consider the address from the previous examples, 1000:1F00. There are several different memory addresses that refer to the same physical address. For example, 11F0:0, 1100:F00, and even 1080:1700 all correspond to physical address 11F00h. When working with certain data types and especially when comparing pointers, it's convenient if segmented addresses point at different objects in memory when their bit representations are different. Clearly this is not always the case in real mode on an 80x86 processor.

Fortunately, there is an easy way to avoid this problem. If you need to compare two addresses for (in)equality, you can use *normalized* addresses. Normalized addresses take a special form so they are all unique. That is, unless two normalized segmented values are exactly the same, they do not point at the same object in memory.

There are many different ways (16, in fact) to create normalized addresses. By convention, most programmers (and high level languages) define a normalized address as follows:

- The segment portion of the address may be any 16 bit value.
- The offset portion must be a value in the range 0..0Fh.

Normalized pointers that take this form are very easy to convert to a physical address. All you need to do is append the single hexadecimal digit of the offset to the segment value. The normalized form of 1000:1F00 is 11F0:0. You can obtain the physical address by appending the offset (zero) to the end of 11F0 yielding 11F00.

It is very easy to convert an arbitrary segmented value to a normalized address. First, convert your segmented address to a physical address using the "multiply by 16 and add in the offset" function. Then slap a colon between the last two digits of the five-digit result:

1000:1F00 ⇒ 11F00 ⇒ 11F0:0

Note that this discussion applies only to 80x86 processors operating in real mode. In protected mode there is no direct correspondence between segmented addresses and physical addresses so this technique does not work. However, this text deals mainly with programs that run in real mode, so normalized pointers appear throughout this text.

4.5 Segment Registers on the 80x86

When Intel designed the 8086 in 1976, memory was a precious commodity. They designed their instruction set so that each instruction would use as few bytes as possible. This made their programs smaller so computer systems employing Intel processors would use less memory. As such, those computer systems cost less to produce. Of course, the cost of memory has plummeted to the point where this is no longer a concern but it was a concern back then[4]. One thing Intel wanted to avoid was appending a 32 bit address (segment:offset) to the end of instructions that reference memory. They were able to reduce this to 16 bits (offset only) by making certain assumptions about which segments in memory an instruction could access.

The 8086 through 80286 processors have four segment registers: cs, ds, ss and es. The 80386 and later processors have these segment registers plus fs and gs. The cs (code segment) register points at the segment containing the currently executing code. The CPU always fetches instructions from the address given by cs:ip. By default, the CPU expects to access most variables in the data segment. Certain variables and other operations occur in the stack segment. When accessing data in these specific areas, no segment value is necessary. To access data in one of the extra segments (es, fs, or gs), only a single byte is necessary to choose the appropriate segment register. Only a few control transfer instructions allow you to specify a full 32 bit segmented address.

Now, this might seem rather limiting. After all, with only four segment registers on the 8086 you can address a maximum of 256 Kilobytes (64K per segment), not the full megabyte promised. However, you can change the segment registers under program control, so it is possible to address any byte by changing the value in a segment register.

Of course, it takes a couple of instructions to change the value of one of the 80x86's segment registers. These instructions consume memory and take time to execute. So saving two bytes per memory access would not pay off if you are accessing data in different segments all the time. Fortunately, most consecutive memory accesses occur in the same segment. Hence, loading segment registers isn't something you do very often.

4.6 The 80x86 Addressing Modes

Like the x86 processors described in the previous chapter, the 80x86 processors let you access memory in many different ways. The 80x86 memory addressing modes provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 80x86 addressing modes is the first step towards mastering 80x86 assembly language.

When Intel designed the original 8086 processor, they provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. Note that the 80386 retained all the modes of the previous processors; the new modes are just an added bonus. If you need to write code that works on 80286 and earlier processors, you will not be able to take advantage of these new modes. However, if you intend to run your code on 80386sx or higher processors, you can use these new modes. Since many programmers still need to write programs that run on 80286 and earlier machines[5], it's important to separate the discussion of these two sets of addressing modes to avoid confusing them.

4. Actually, small programs are still important. The smaller a program is the faster it will run because the CPU has to fetch fewer bytes from memory and the instructions don't take up as much of the cache.

5. Modern PCs rarely use processors earlier than the 80386, but embedded system still use the older processors.

4.6.1 8086 Register Addressing Modes

Most 8086 instructions can operate on the 8086's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 8086 mov (move) instruction:

```
mov     destination, source
```

This instruction copies the data from the *source* operand to the *destination* operand. The eight and 16 bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 8086 mov instructions:

```
mov     ax, bx      ;Copies the value from BX into AX
mov     dl, al      ;Copies the value from AL into DL
mov     si, dx      ;Copies the value from DX into SI
mov     sp, bp      ;Copies the value from BP into SP
mov     dh, cl      ;Copies the value from CL into DH
mov     ax, ax      ;Yes, this is legal!
```

Remember, the registers are the best place to keep often used variables. As you'll see a little later, instructions using the registers are shorter and faster than those that access memory. Throughout this chapter you'll see the abbreviated operands *reg* and *r/m* (register / memory) used wherever you may use one of the 8086's general purpose registers.

In addition to the general purpose registers, many 8086 instructions (including the mov instruction) allow you to specify one of the segment registers as an operand. There are two restrictions on the use of the segment registers with the mov instruction. First of all, you may not specify cs as the destination operand, second, only one of the operands can be a segment register. You cannot move data from one segment register to another with a single mov instruction. To copy the value of cs to ds, you'd have to use some sequence like:

```
mov     ax, cs
mov     ds, ax
```

You should never use the segment registers as data registers to hold arbitrary values. They should only contain segment addresses. But more on that, later. Throughout this text you'll see the abbreviated operand *sreg* used wherever segment register operands are allowed (or required).

4.6.2 8086 Memory Addressing Modes

The 8086 provides 17 different ways to access memory. This may seem like quite a bit at first[6], but fortunately most of the address modes are simple variants of one another so they're very easy to learn. And learn them you should! The key to good assembly language programming is the proper use of memory addressing modes.

The addressing modes provided by the 8086 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed. Variations on these five forms provide the 17 different addressing modes on the 8086. See, from 17 down to five. It's not so bad after all!

4.6.2.1 The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction mov al,ds:[8088h] loads the al register with a copy of the byte at memory loca-

6. Just wait until you see the 80386!

MASM Syntax for 8086 Memory Addressing Modes

Microsoft's assembler uses several different variations to denote indexed, based/indexed, and displacement plus based/indexed addressing modes. You will see all of these forms used interchangeably throughout this text. The following list some of the possible combinations that are legal for the various 80x86 addressing modes:

```
disp[bx], [bx][disp], [bx+disp], [disp][bx], and [disp+bx]

[bx][si], [bx+si], [si][bx], and [si+bx]

disp[bx][si], disp[bx+si], [disp+bx+si], [disp+bx][si], disp[si][bx], [disp+si][bx],
[disp+si+bx], [si+disp+bx], [bx+disp+si], etc.
```

MASM treats the "[ ]" symbols just like the "+" operator. This operator is commutative, just like the "+" operator. Of course, this discussion applies to all the 8086 addressing modes, not just those involving BX and SI. You may substitute any legal registers in the addressing modes above.

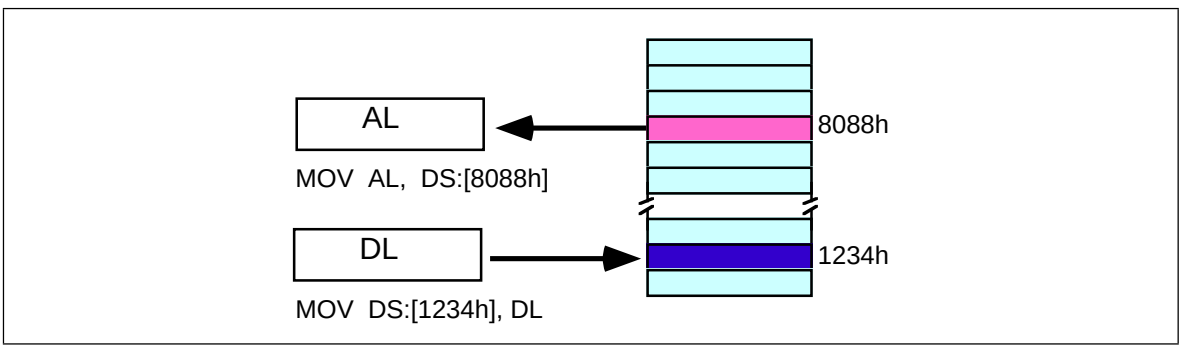


Figure 4.8 Displacement Only (Direct) Addressing Mode

tion 8088h[7]. Likewise, the instruction mov ds:[1234h],dl stores the value in the dl register to memory location 1234h (see Figure 4.8)

The displacement-only addressing mode is perfect for accessing simple variables. Of course, you'd probably prefer using names like "I" or "J" rather than "DS:[1234h]" or "DS:[8088h]". Well, fear not, you'll soon see it's possible to do just that.

Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the mov opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors (see the previous chapter). There are some minor differences, however. First of all, a displacement is exactly that– some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). Don't worry if this doesn't make a lot of sense right now. You'll get an opportunity to study segments to your heart's content a little later in this chapter. For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words on the 8086 processors[8] (see Figure 4.9).

By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a *segment override prefix* before your address. For example, to access location 1234h in the extra segment (es) you would use an instruction of the form mov ax,es:[1234h]. Likewise, to access this location in the code segment you would use the instruction mov ax, cs:[1234h]. The ds: prefix in the previous examples is *not* a segment override. The CPU uses the data segment register by default. These specific examples require ds: because of MASM's syntactical limitations.

7. The purpose of the "DS:" prefix on the instruction will become clear a little later.
8. And double words on the 80386 and later.

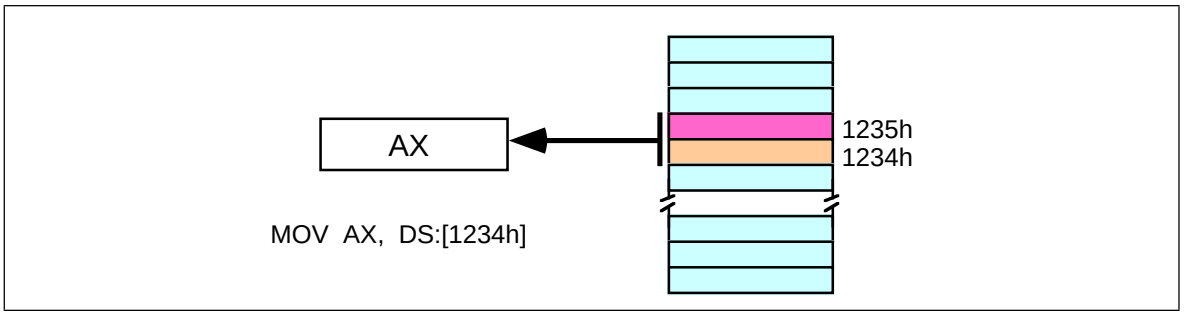


Figure 4.9 Accessing a Word

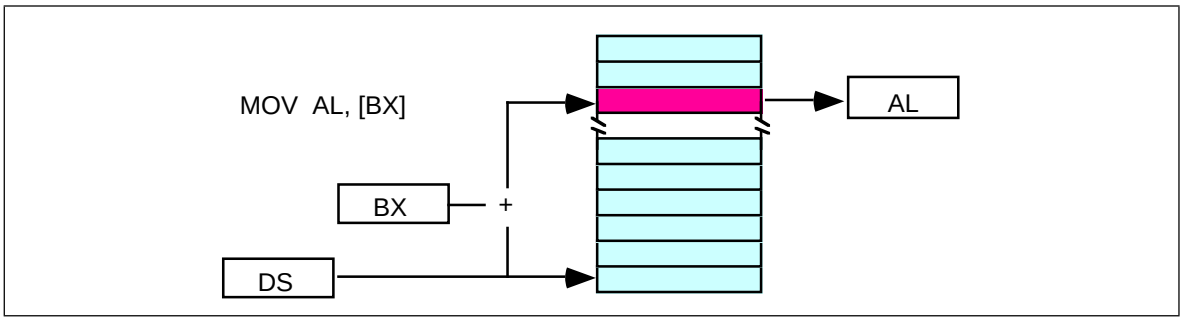


Figure 4.10 [BX] Addressing Mode

4.6.2.2 The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

As with the x86 [bx] addressing mode, these four addressing modes reference the byte at the offset found in the bx, bp, si, or di register, respectively. The [bx], [si], and [di] modes use the ds segment by default. The [bp] addressing mode uses the stack segment (ss) by default.

You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
mov     al, cs:[bx]
mov     al, ds:[bp]
mov     al, ss:[si]
mov     al, es:[di]
```

Intel refers to [bx] and [bp] as *base addressing modes* and bx and bp as *base registers* (in fact, bp stands for base pointer). Intel refers to the [si] and [di] addressing modes as *indexed addressing modes* (si stands for *source index*, di stands for *destination index*). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.

Note: the [si] and [di] addressing modes work exactly the same way, just substitute si and di for bx above.

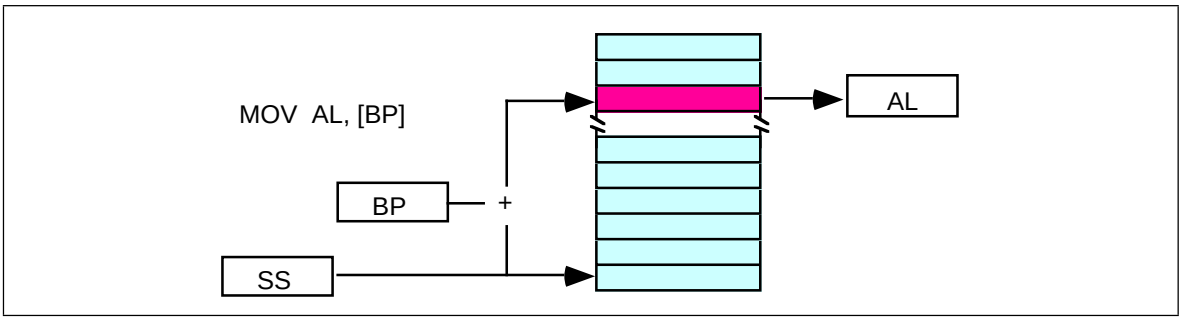


Figure 4.11 [BP] Addressing Mode

4.6.2.3 Indexed Addressing Modes

The indexed addressing modes use the following syntax:

```
mov     al, disp[bx]
mov     al, disp[bp]
mov     al, disp[si]
mov     al, disp[di]
```

If bx contains 1000h, then the instruction mov cl,20h[bx] will load cl from memory location ds:1020h. Likewise, if bp contains 2020h, mov dh,1000h[bp] will load dh from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving bx, si, and di all use the data segment, the disp[bp] addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

```
mov     al, ss:disp[bx]
mov     al, es:disp[bp]
mov     al, cs:disp[si]
mov     al, ss:disp[di]
```

You may substitute si or di in Figure 4.12 to obtain the [si+disp] and [di+disp] addressing modes.

Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel's literature does not differentiate between these modes with or without the constant. If you look at how the hardware works, this is a reasonable definition. From the programmer's point of view, however, these addressing modes are useful for entirely

Based vs. Indexed Addressing

There is actually a subtle difference between the based and indexed addressing modes. Both addressing modes consist of a displacement added together with a register. The major difference between the two is the relative sizes of the displacement and register values. In the indexed addressing mode, the constant typically provides the address of the specific data structure and the register provides an offset from that address. In the based addressing mode, the register contains the address of the data structure and the constant displacement supplies the index from that point.

Since addition is commutative, the two views are essentially equivalent. However, since Intel supports one and two byte displacements (See "The 80x86 MOV Instruction" on page 166) it made more sense for them to call it the based addressing mode. In actual use, however, you'll wind up using it as an indexed addressing mode more often than as a based addressing mode, hence the name change.

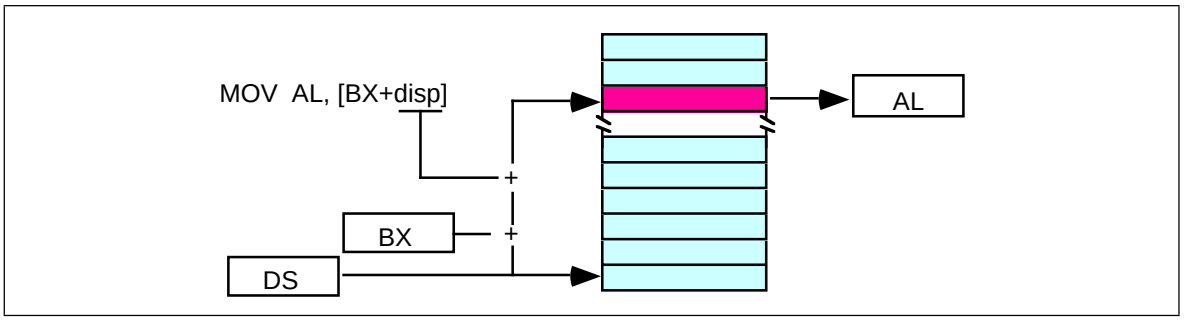


Figure 4.12 [BX+disp] Addressing Mode

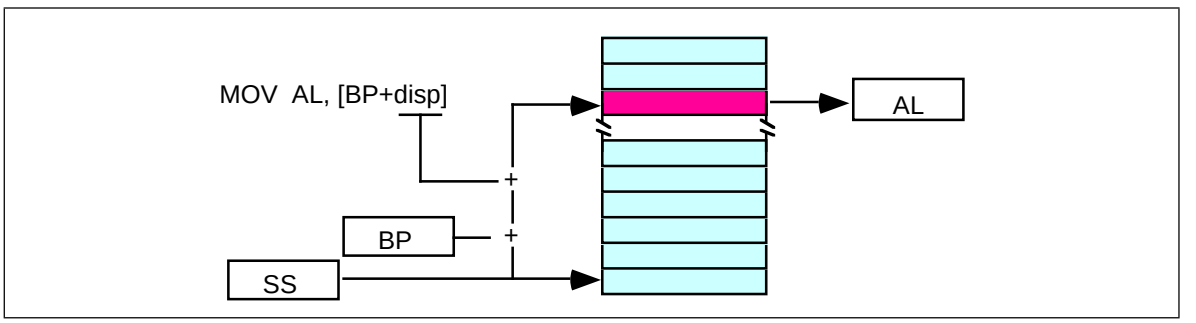


Figure 4.13 [BP+disp] Addressing Mode

different things. Which is why this text uses different terms to describe them. Unfortunately, there is very little consensus on the use of these terms in the 80x86 world.

4.6.2.4 Based Indexed Addressing Modes

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

```
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
```

Suppose that bx contains 1000h and si contains 880h. Then the instruction

```
mov     al,[bx][si]
```

would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, mov ax,[bp+di] will load the 16 bits in ax from locations SS:259C and SS:259D.

The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.

You substitute di in Figure 4.12 to obtain the [bx+di] addressing mode. You substitute di in Figure 4.12 for the [bp+di] addressing mode.

4.6.2.5 Based Indexed Plus Displacement Addressing Mode

These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes (see Figure 4.12 and Figure 4.12).

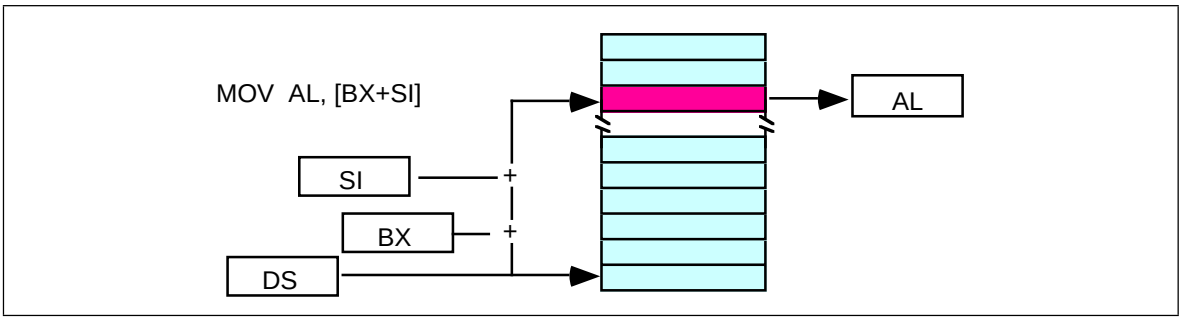


Figure 4.14 [BX+SI] Addressing Mode

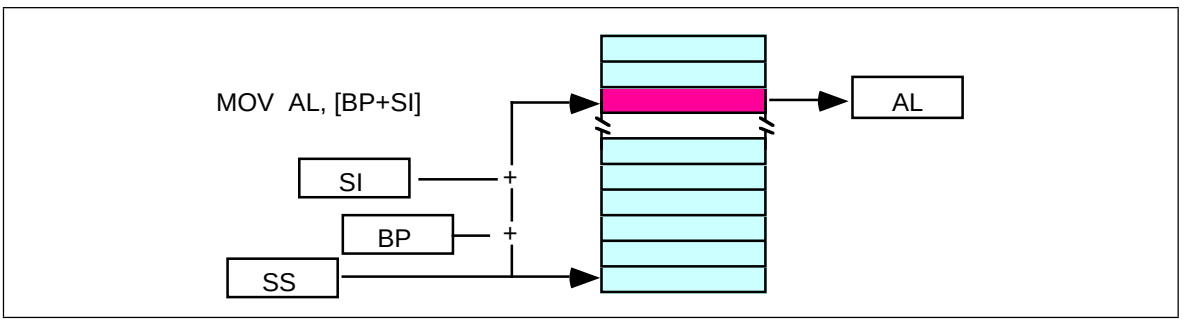


Figure 4.15 [BP+SI] Addressing Mode

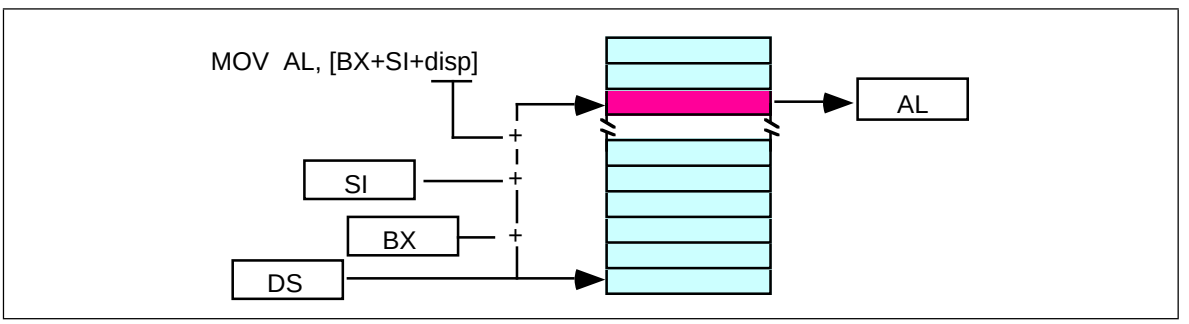


Figure 4.16 [BX + SI + disp] Addressing Mode

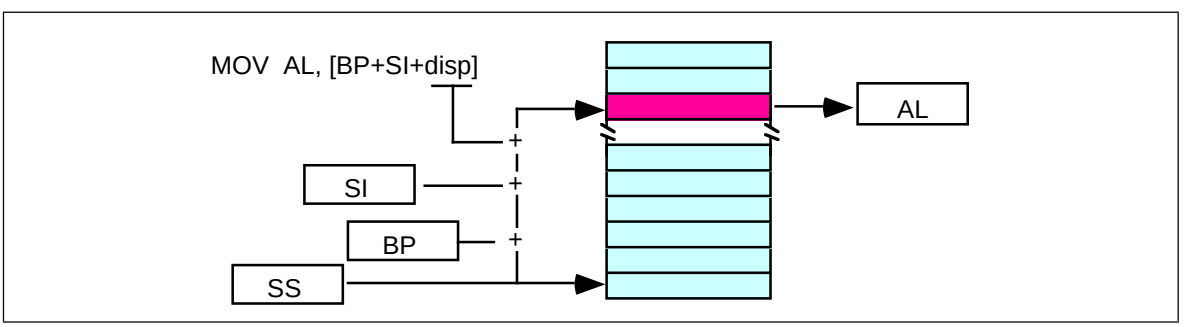


Figure 4.17 [BP + SI + disp] Addressing Mode

```
mov       al, disp[bx][si]
mov       al, disp[bx+di]
mov       al, [bp+si+disp]
mov       al, [bp][di][disp]
```

You may substitute di in Figure 4.12 to produce the [bx+di+disp] addressing mode. You may substitute di in Figure 4.12 to produce the [bp+di+disp] addressing mode.

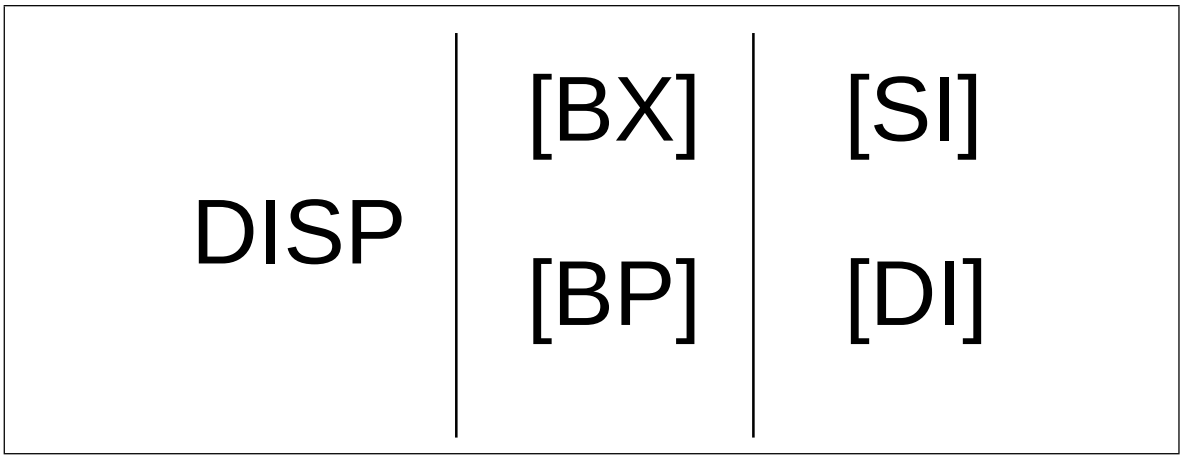


Figure 4.18 Table to Generate Valid 8086 Addressing Modes

Suppose bp contains 1000h, bx contains 2000h, si contains 120h, and di contains 5. Then mov al,10h[bx+si] loads al from address DS:2130; mov ch,125h[bp+di] loads ch from location SS:112A; and mov bx,cs:2[bx][di] loads bx from location CS:2007.

4.6.2.6 An Easy Way to Remember the 8086 Memory Addressing Modes

There are a total of 17 different legal memory addressing modes on the 8086: disp, [bx], [bp], [si], [di], disp[bx], disp[bp], disp[si], disp[di], [bx][si], [bx][di], [bp][si], [bp][di], disp[bx][si], disp [bx][di], disp[bp][si], and disp[bp][di][9]. You could memorize all these forms so that you know which are valid (and, by omission, which forms are invalid). However, there is an easier way besides memorizing these 17 forms. Consider the chart in Figure 4.12.

If you choose zero or one items from each of the columns and wind up with at least one item, you've got a valid 8086 memory addressing mode. Some examples:

- Choose disp from column one, nothing from column two, [di] from column 3, you get disp[di].
- Choose disp, [bx], and [di]. You get disp[bx][di].
- Skip column one & two, choose [si]. You get [si]
- Skip column one, choose [bx], then choose [di]. You get [bx][di]

Likewise, if you have an addressing mode that you *cannot* construct from this table, then it is not legal. For example, disp[dx][si] is illegal because you cannot obtain [dx] from any of the columns above.

4.6.2.7 Some Final Comments About 8086 Addressing Modes

The *effective address* is the final offset produced by an addressing mode computation. For example, if bx contains 10h, the effective address for 10h[bx] is 20h. You will see the term effective address in almost any discussion of the 8086's addressing mode. There is even a special instruction *load effective address* (lea) that computes effective addresses.

Not all addressing modes are created equal! Different addressing modes may take differing amounts of time to compute the effective address. The exact difference varies from processor to processor. Generally, though, the more complex an addressing mode is, the longer it takes to compute the effective address. Complexity of an addressing mode is directly related to the number of terms in the addressing mode. For example, disp[bx][si] is

9. That's not even counting the syntactical variations!

more complex than [bx]. See the instruction set reference in the appendices for information regarding the cycle times of various addressing modes on the different 80x86 processors.

The displacement field in all addressing modes *except* displacement-only can be a signed eight bit constant or a signed 16 bit constant. If your offset is in the range -128…+127 the instruction will be shorter (and therefore faster) than an instruction with a displacement outside that range. The size of the value in the register does not affect the execution time or size. So if you can arrange to put a large number in the register(s) and use a small displacement, that is preferable over a large constant and small values in the register(s).

If the effective address calculation produces a value greater than 0FFFFh, the CPU ignores the overflow and the result *wraps around* back to zero. For example, if bx contains 10h, then the instruction mov al,0FFFFh[bx] will load the al register from location ds:0Fh, not from location ds:1000Fh.

In this discussion you've seen how these addressing modes operate. The preceding discussion didn't explain *what you use them for.* That will come a little later. As long as you know how each addressing mode performs its effective address calculation, you'll be fine.

4.6.3 80386 Register Addressing Modes

The 80386 (and later) processors provide 32 bit registers. The eight general-purpose registers all have 32 bit equivalents. They are eax, ebx, ecx, edx, esi, edi, ebp, and esp. If you are using an 80386 or later processor you can use these registers as operands to several 80386 instructions.

4.6.4 80386 Memory Addressing Modes

The 80386 processor generalized the memory addressing modes. Whereas the 8086 only allowed you to use bx or bp as base registers and si or di as index registers, the 80386 lets you use almost any general purpose 32 bit register as a base or index register. Furthermore, the 80386 introduced new *scaled indexed* addressing modes that simplify accessing elements of arrays. Beyond the increase to 32 bits, the new addressing modes on the 80386 are probably the biggest improvement to the chip over earlier processors.

4.6.4.1 Register Indirect Addressing Modes

On the 80386 you may specify *any* general purpose 32 bit register when using the register indirect addressing mode. [eax], [ebx], [ecx], [edx], [esi], and [edi] all provide offsets, by default, into the data segment. The [ebp] and [esp] addressing modes use the stack segment by default.

Note that while running in 16 bit real mode on the 80386, offsets in these 32 bit registers must still be in the range 0…0FFFFh. You cannot use values larger than this to access more than 64K in a segment[10]. Also note that you must use the 32 bit names of the registers. You cannot use the 16 bit names. The following instructions demonstrate all the legal forms:

```
mov     al, [eax]
mov     al, [ebx]
mov     al, [ecx]
mov     al, [edx]
mov     al, [esi]
mov     al, [edi]
mov     al, [ebp]    ;Uses SS by default.
```

10. Unless, of course, you're operating in protected mode, in which case this is perfectly legal.

```
mov     al, [esp]    ;Uses SS by default.
```

4.6.4.2 80386 Indexed, Base/Indexed, and Base/Indexed/Disp Addressing Modes

The indexed addressing modes (register indirect plus a displacement) allow you to mix a 32 bit register with a constant. The base/indexed addressing modes let you pair up two 32 bit registers. Finally, the base/indexed/displacement addressing modes let you combine a constant and two registers to form the effective address. Keep in mind that the offset produced by the effective address computation must still be 16 bits long when operating in real mode.

On the 80386 the terms *base register* and *index register* actually take on some meaning. When combining two 32 bit registers in an addressing mode, the first register is the base register and the second register is the index register. This is true regardless of the register names. Note that the 80386 allows you to use the *same* register as both a base and index register, which is actually useful on occasion. The following instructions provide representative samples of the various base and indexed address modes along with syntactical variations:

```
mov     al, disp[eax]          ;Indexed addressing
mov     al, [ebx+disp]         ; modes.
mov     al, [ecx][disp]
mov     al, disp[edx]
mov     al, disp[esi]
mov     al, disp[edi]
mov     al, disp[ebp]          ;Uses SS by default.
mov     al, disp[esp]          ;Uses SS by default.
```

The following instructions all use the base+indexed addressing mode. The first register in the second operand is the base register, the second is the index register. If the *base* register is esp or ebp the effective address is relative to the stack segment. Otherwise the effective address is relative to the data segment. Note that the choice of index register does not affect the choice of the default segment.

```
mov     al, [eax][ebx]         ;Base+indexed addressing
mov     al, [ebx+ebx]          ; modes.
mov     al, [ecx][edx]
mov     al, [edx][ebp]         ;Uses DS by default.
mov     al, [esi][edi]
mov     al, [edi][esi]
mov     al, [ebp+ebx]          ;Uses SS by default.
mov     al, [esp][ecx]         ;Uses SS by default.
```

Naturally, you can add a displacement to the above addressing modes to produce the base+indexed+displacement addressing mode. The following instructions provide a representative sample of the possible addressing modes:

```
mov     al, disp[eax][ebx]     ;Base+indexed addressing
mov     al, disp[ebx+ebx]      ; modes.
mov     al, [ecx+edx+disp]
mov     al, disp[edx+ebp]      ;Uses DS by default.
mov     al, [esi][edi][disp]
mov     al, [edi][disp][esi]
mov     al, disp[ebp+ebx]      ;Uses SS by default.
mov     al, [esp+ecx][disp]    ;Uses SS by default.
```

There is one restriction the 80386 places on the index register. You cannot use the esp register as an index register. It's okay to use esp as the base register, but not as the index register.

4.6.4.3 80386 Scaled Indexed Addressing Modes

The indexed, base/indexed, and base/indexed/disp addressing modes described above are really special instances of the 80386 *scaled indexed addressing modes*. These addressing modes are particularly useful for accessing elements of arrays, though they are not limited to such purposes. These modes let you multiply the index register in the addressing mode by one, two, four, or eight. The general syntax for these addressing modes is

```
disp[index*n]
[base][index*n]
```

or

```
disp[base][index*n]
```

where "base" and "index" represent any 80386 32 bit general purpose registers and "n" is the value one, two, four, or eight.

The 80386 computes the effective address by adding disp, base, and index*n together. For example, if ebx contains 1000h and esi contains 4, then

```
mov al,8[ebx][esi*4]            ;Loads AL from location 1018h
mov al,1000h[ebx][ebx*2]        ;Loads AL from location 4000h
mov al,1000h[esi*8]             ;Loads AL from location 1020h
```

Note that the 80386 extended indexed, base/indexed, and base/indexed/displacement addressing modes really are special cases of this scaled indexed addressing mode with "n" equal to one. That is, the following pairs of instructions are absolutely identical to the 80386:

```
mov    al, 2[ebx][esi*1]          mov       al, 2[ebx][esi]
mov    al, [ebx][esi*1]           mov       al, [ebx][esi]
mov    al, 2[esi*1]               mov       al, 2[esi]
```

Of course, MASM allows lots of different variations on these addressing modes. The following provide a small sampling of the possibilities:

```
disp[bx][si*2], [bx+disp][si*2], [bx+si*2+disp], [si*2+bx][disp],
disp[si*2][bx], [si*2+disp][bx], [disp+bx][si*2]
```

4.6.4.4 Some Final Notes About the 80386 Memory Addressing Modes

Because the 80386's addressing modes are more orthogonal, they are much easier to memorize than the 8086's addressing modes. For programmers working on the 80386 processor, there is always the temptation to skip the 8086 addressing modes and use the 80386 set exclusively. However, as you'll see in the next section, the 8086 addressing modes really are more efficient than the comparable 80386 addressing modes. Therefore, it is important that you know *all* the addressing modes and choose the mode appropriate to the problem at hand.

When using base/indexed and base/indexed/disp addressing modes on the 80386, without a scaling option (that is, letting the scaling default to "*1"), the first register appearing in the addressing mode is the base register and the second is the index register. This is an important point because the choice of the default segment is made by the choice of the base register. If the base register is ebp or esp, the 80386 defaults to the stack segment. In all other cases the 80386 accesses the data segment by default, *even if the index register is ebp*. If you use the scaled index operator ("*n") on a register, that register is always the index register regardless of where it appears in the addressing mode:

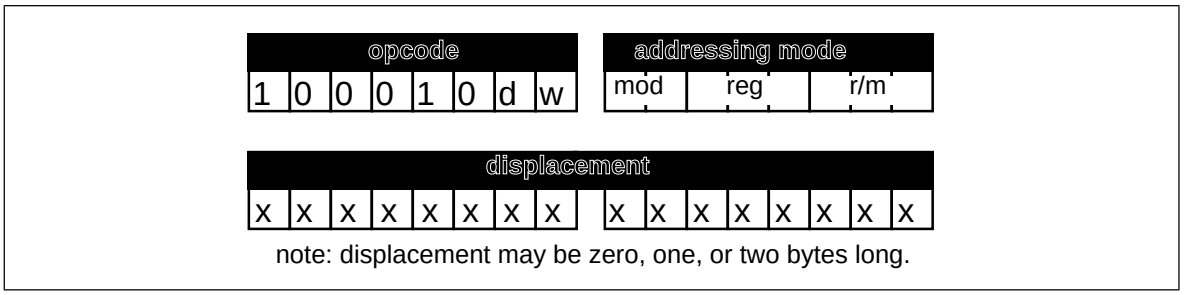


Figure 4.19 Generic MOV Instruction

```
[ebx][ebp]          ;Uses DS by default.
[ebp][ebx]          ;Uses SS by default.
[ebp*1][ebx]        ;Uses DS by default.
[ebx][ebp*1]        ;Uses DS by default.
[ebp][ebx*1]        ;Uses SS by default.
[ebx*1][ebp]        ;Uses SS by default.
es:[ebx][ebp*1]     ;Uses ES.
```

4.7 The 80x86 MOV Instruction

The examples throughout this chapter will make extensive use of the 80x86 mov (move) instruction. Furthermore, the mov instruction is the most common 80x86 machine instruction. Therefore, it's worthwhile to spend a few moments discussing the operation of this instruction.

Like it's x86 counterpart, the mov instruction is very simple. It takes the form:

```
mov     Dest,Source
```

Mov makes a copy of *Source* and stores this value into *Dest*. This instruction does not affect the original contents of *Source*. It overwrites the previous value in *Dest*. For the most part, the operation of this instruction is completely described by the Pascal statement:

```
Dest := Source;
```

This instruction has many limitations. You'll get ample opportunity to deal with them throughout your study of 80x86 assembly language. To understand why these limitations exist, you're going to have to take a look at the machine code for the various forms of this instruction. One word of warning, they don't call the 80386 a CISC (Complex Instruction Set Computer) for nothing. The encoding for the `mov` instruction is probably the most complex in the instruction set. Nonetheless, without studying the machine code for this instruction you will not be able to appreciate it, nor will you have a good understanding of how to write optimal code using this instruction. You'll see why you worked with the x86 processors in the previous chapters rather than using actual 80x86 instructions.

There are several versions of the mov instruction. The mnemonic[11] mov describes over a dozen different instructions on the 80386. The most commonly used form of the mov instruction has the following binary encoding shown in Figure 4.19.

The opcode is the first eight bits of the instruction. Bits zero and one define the *width* of the instruction (8, 16, or 32 bits) and the *direction* of the transfer. When discussing specific instructions this text will always fill in the values of *d* and *w* for you. They appear here only because almost every other text on this subject requires that *you* fill in these values.

Following the opcode is the addressing mode byte, affectionately called the "mod-reg-r/m" byte by most programmers. This byte chooses which of 256 different pos-

11. Mnemonic means *memory aid*. This term describes the English names for instructions like MOV, ADD, SUB, etc., which are much easier to remember than the hexadecimal encodings for the machine instructions.

sible operand combinations the generic mov instruction allows. The generic mov instruction takes three different assembly language forms:

```
mov       reg, memory
mov       memory, reg
mov       reg, reg
```

Note that at least one of the operands is always a general purpose register. The *reg* field in the mod/reg/rm byte specifies that register (or one of the registers if using the third form above). The *d* (direction) bit in the opcode decides whether the instruction stores data into the register (d=1) or into memory (d=0).

The bits in the reg field let you select one of eight different registers. The 8086 supports 8 eight bit registers and 8 sixteen bit general purpose registers. The 80386 also supports eight 32 bit general purpose registers. The CPU decodes the meaning of the *reg* field as follows:

Table 23: REG Bit Encodings

reg	w=0	16 bit mode w=1	32 bit mode w=1
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

To differentiate 16 and 32 bit register, the 80386 and later processors use a special opcode prefix byte before instructions using the 32 bit registers. Otherwise, the instruction encodings are the same for both types of instructions.

The r/m field, in conjunction with the mod field, chooses the addressing mode. The mod field encoding is the following:

Table 24: MOD Encoding

MOD	Meaning
00	The r/m field denotes a register indirect memory addressing mode or a base/indexed addressing mode (see the encodings for r/m) *unless* the r/m field contains 110. If MOD=00 and r/m=110 the mod and r/m fields denote displacement-only (direct) addressing.
01	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is an eight bit signed displacement following the mod/reg/rm byte.
10	The r/m field denotes an indexed or base/indexed/displacement addressing mode. There is a 16 bit signed displacement (in 16 bit mode) or a 32 bit signed displacement (in 32 bit mode) following the mod/reg/rm byte .
11	The r/m field denotes a register and uses the same encoding as the *reg* field

The mod field chooses between a register-to-register move and a register-to/from-memory move. It also chooses the size of the displacement (zero, one, two, or four bytes) that follows the instruction for memory addressing modes. If MOD=00, then you have one of the addressing modes without a displacement (register indirect or base/indexed). Note the special case where MOD=00 and r/m=110. This would normally correspond to the [bp]

addressing mode. The 8086 uses this encoding for the displacement-only addressing mode. This means that *there isn't a true [bp] addressing mode on the 8086.*

To understand why you can use the [bp] addressing mode in your programs, look at MOD=01 and MOD=10 in the above table. These bit patterns activate the disp[reg] and the disp[reg][reg] addressing modes. "So what?" you say. "That's not the same as the [bp] addressing mode." And you're right. However, consider the following instructions:

```
mov     al, 0[bx]
mov     ah, 0[bp]
mov     0[si], al
mov     0[di], ah
```

These statements, using the indexed addressing modes, perform the same operations as their register indirect counterparts (obtained by removing the displacement from the above instructions). The only real difference between the two forms is that the indexed addressing mode is one byte longer (if MOD=01, two bytes longer if MOD=10) to hold the displacement of zero. Because they are longer, these instructions may also run a little slower.

This trait of the 8086 – providing two or more ways to accomplish the same thing – appears throughout the instruction set. In fact, you're going to see several more examples before you're through with the mov instruction. MASM generally picks the best form of the instruction automatically. Were you to enter the code above and assemble it using MASM, it would still generate the register indirect addressing mode for all the instructions except mov ah,0[bp]. It would, however, emit only a one-byte displacement that is shorter and faster than the same instruction with a two-byte displacement of zero. Note that MASM does not require that you enter 0[bp], you can enter [bp] and MASM will automatically supply the zero byte for you.

If MOD does not equal 11b, the r/m field encodes the memory addressing mode as follows:

Table 25: R/M Field Encoding

R/M	Addressing mode (Assuming MOD=00, 01, or 10)
000	[BX+SI] or DISP[BX][SI] (depends on MOD)
001	[BX+DI] or DISP[BX+DI] (depends on MOD)
010	[BP+SI] or DISP[BP+SI] (depends on MOD)
011	[BP+DI] or DISP[BP+DI] (depends on MOD)
100	[SI] or DISP[SI] (depends on MOD)
101	[DI] or DISP[DI] (depends on MOD)
110	Displacement-only or DISP[BP] (depends on MOD)
111	[BX] or DISP[BX] (depends on MOD)

Don't forget that addressing modes involving bp use the stack segment (ss) by default. All others use the data segment (ds) by default.

If this discussion has got you totally lost, you haven't even seen the worst of it yet. Keep in mind, these are just *some* of the 8086 addressing modes. *You've still got all the 80386 addressing modes to look at.* You're probably beginning to understand what they mean when they say *complex* instruction set computer. However, the important concept to note is that you can construct 80x86 instructions the same way you constructed x86 instructions in Chapter Three – by building up the instruction bit by bit. For full details on how the 80x86 encodes instructions, see the appendices.

4.8 Some Final Comments on the MOV Instructions

There are several important facts you should always remember about the mov instruction. First of all, *there are no memory to memory moves*. For some reason, newcomers to assembly language have a hard time grasping this point. While there are a couple of instructions that perform memory to memory moves, loading a register and then storing that register is almost always more efficient. Another important fact to remember about the mov instruction is that there are many different mov instructions that accomplish the same thing. Likewise, there are several different addressing modes you can use to access the same memory location. If you are interested in writing the shortest and fastest possible programs in assembly language, you must be constantly aware of the trade-offs between equivalent instructions.

The discussion in this chapter deals mainly with the generic mov instruction so you can see how the 80x86 processors encode the memory and register addressing modes into the mov instruction. Other forms of the mov instruction let you transfer data between 16-bit general purpose registers and the 80x86 segment registers. Others let you load a register or memory location with a constant. These variants of the mov instruction use a different opcode. For more details, see the instruction encodings in Appendix D.

There are several additional mov instructions on the 80386 that let you load the 80386 special purpose registers. This text will not consider them. There are also some string instructions on the 80x86 that perform memory to memory operations. Such instructions appear in the next chapter. They are not a good substitute for the mov instruction.

4.9 Laboratory Exercises

It is now time to begin working with actual 80x86 assembly language. To do so, you will need to learn how to use several assembly-language related software development tools. In this set of laboratory exercises you will learn how to use the basic tools to edit, assemble, debug, and run 80x86 assembly language programs. These exercises assume that you have already installed MASM (Microsoft's Macro Assembler) on your system. If you have not done so already, please install MASM (following Microsoft's directions) before attempting the exercises in this laboratory.

4.9.1 The UCR Standard Library for 80x86 Assembly Language Programmers

Most of the programs in this textbook use a set of standard library routines created at the University of California, Riverside. These routines provide standardized I/O, string handling, arithmetic, and other useful functions. The library itself is very similar to the C standard library commonly used by C/C++ programmers. Later chapters in this text will describe many of the routines found in the library, there is no need to go into that here. However, many of the example programs in this chapter and in later chapters will use certain library routines, so you must install and activate the library at this time.

The library appears on the companion CD-ROM. You will need to copy the library from CD-ROM to the hard disk. A set of commands like the following (with appropriate adjustments for the CD-ROM drive letter) will do the trick:

```
c:
cd \
md stdlib
cd stdlib
xcopy r:\stdlib\*.* . /s
```

Once you've copied the library to your hard disk, there are two additional commands you must execute before attempting to assemble any code that uses the standard library:

```
set include=c:\stdlib\include
set lib=c:\stdlib\lib
```

It would probably be a good idea to place these commands in your autoexec.bat file so they execute automatically every time you start up your system. If you have not set the include and lib variables, MASM will complain during assembly about missing files.

4.9.2 Editing Your Source Files

Before you can assemble (compile) and run your program, you must create an assembly language source file with an editor. MASM will properly handle any ASCII text file, so it doesn't matter what editor you use to create that file as long as that editor processes ASCII text files. Note that most word processors *do not* normally work with ASCII text files, therefore, you should not use a word processor to maintain your assembly language source files.

MS-DOS, Windows, and MASM all three come with simple text editors you can use to create and modify assembly language source files. The EDIT.EXE program comes with MS-DOS; The NOTEPAD.EXE application comes with Windows; and the PWB (Programmer's Work Bench) comes with MASM. If you do not have a favorite text editor, feel free to use one of these programs to edit your source code. If you have some language system (e.g., Borland C++, Delphi, or MS Visual C++) you can use the editor they provide to prepare your assembly language programs, if you prefer.

Given the wide variety of possible editors out there, this chapter will not attempt to describe how to use any of them. If you've never used a text editor on the PC before, consult the appropriate documentation for that text editor.

4.9.3 The SHELL.ASM File

Although you can write an assembly language program completely from scratch within your text editor of choice, most assembly language programs contain a large number of statements common to every assembly language program. In the Chapter Four directory on the companion CD-ROM there is a "SHELL.ASM" text file. The SHELL.ASM file is a skeleton assembly language file[12]. That is, it contains all the "overhead" instructions necessary to create a working assembly language program with the exception of the instructions and variables that make up that specific program. In many respects, it is comparable to the following Pascal program:

```
program shell(input,output);
begin
end.
```

Which is to say that SHELL.ASM is a valid program. You can assemble and run it but it won't do very much.

The main reason for the SHELL.ASM program is that there are lots of lines of code that must appear in an empty assembly language program just to make the assembler happy. Unfortunately, to understand what these instructions mean requires considerable study. Rather than put off writing any programs until you understand everything necessary to create your first program, you're going to blindly use the SHELL.ASM file without questioning what any of it means. Fear not. Within a couple chapters it will all make sense. But for now, just type it in and use it exactly as it appears. The only thing you need to know about SHELL.ASM right away is where to place your code in this file. That's easy to see, though; there are three comments in the file telling you where to put your variables (if any), subroutine / procedures / functions (if any), and the statements for your main pro-

12. This file is available on the companion CD-ROM.

gram. The following is the complete listing of the SHELL.ASM file for those who may not have access to the electronic version:

```
                        .xlist
                        include     stdlib.a
                        includelib  stdlib.lib
                        .list

dseg                    segment     para public 'data'

; Global variables go here:

dseg                    ends

cseg                    segment     para public 'code'
                        assume      cs:cseg, ds:dseg

; Variables that wind up being used by the standard library routines.
; The MemInit routine uses "PSP" and "zzzzzzseg" labels. They must be
; present if you intend to use getenv, MemInit, malloc, and free.

                        public      PSP
PSP                     dw          ?

;-----------------------------------------
; Here is a good place to put other routines:
;-----------------------------------------------------------------
; Main is the main program. Program execution always begins here.

Main                    proc
                        mov         cs:PSP, es      ;Save pgm seg prefix
                        mov         ax, seg dseg    ;Set up the segment
registers
                        mov         ds, ax
                        mov         es, ax

                        mov         dx, 0
                        meminit
                        jnc         GoodMemInit

                        print
                        db          "Error initializing memory
manager",cr,lf,0
                        jmp         Quit
GoodMemInit:

;*****************************************************************************
; Put your main program here.
;*****************************************************************************

Quit:                   ExitPgm
Main                    endp
cseg                    ends

; Allocate a reasonable amount of space for the stack (2k).

sseg                    segment     para stack 'stack'
stk                     db          256 dup ("stack ")
sseg                    ends

; zzzzzzseg must be the last segment that gets loaded into memory!

zzzzzzseg               segment     para public 'zzzzzz'
LastBytes               db          16 dup (?)
zzzzzzseg               ends
                        end         Main
```

Although you're supposed to simply accept this code as-is and without question, a few explanations are in order. The program itself begins with a pair of "include" and "includelib" statements. These statements tell the assembler and linker that this code will be using some of the library routines from the "UCR Standard Library for 80x86 Assembly Language Programmers." This library appears on the companion CD-ROM.

Note that text beginning with a semicolon (";") is a comment. The assembler ignores all the text from the semicolon to the end of the line. As with high level languages, comments are very important for explaining the operation of your program. In this example, the comments point out some important parts of the SHELL.ASM program[13].

The next section of interest is the line that begins with dseg segment .... This is the beginning of your global data area. This statement defines the beginning of a data segment (*dseg* stands for data segment) that ends with the dseg ends statement. You should place all your global variables between these two statements.

Next comes the code segment (it's called *cseg*) where the 80x86 instructions go. The important thing to note here is the comment "Put your main program here." For now, you should ignore everything else in the code segment except this one comment. The sequences of assembly language statements you create should go between the lines of asterisks surrounding this comment. Have no fear; you'll learn what all these statements mean in the next two chapters. Attempting to explain them now would simply be too much of a digression.

Finally come two additional segments in the program: sseg and zzzzzzseg. These segments are absolutely necessary (the system requires sseg, the UCR Standard Library requires zzzzzzseg). You should not modify these segments.

When you begin writing a new assembly language program you should *not* modify the SHELL.ASM file directly. You should first make a copy of SHELL.ASM using the DOS copy command. For example, you might copy the file to PROJECT1.ASM and then make all your modifications to this file. By doing this you will have an undisturbed copy of SHELL.ASM available for your next project.

There is a special version of SHELL.ASM, X86.ASM, that contains some additional code to support programming projects in this chapter. Please see the programming projects section for more details.

4.9.4 Assembling Your Code with MASM

To run MASM you use the ML.EXE (MASM and Link) program. This file is typically found in a directory with a name like C:\MASM611\BIN. You should check to see if your path includes this directory. If not, you should adjust the DOS shell path variable so that it includes the directory containing ML.EXE, LINK.EXE, CV.EXE, and other MASM-related programs.

MASM is a DOS-based program. The easiest way to run it is from DOS or from a DOS box inside Windows. The basic MASM command takes the following form:

```
ml {options} filename.asm
```

Note that the ML program requires that you type the ".asm" suffix to the filename when assembling an assembly language source file.

Most of the time, you will only use the "/Zi" option. This tells MASM to add symbolic debugging information to the .EXE file for use by CodeView. This makes the executable file somewhat larger, but it also makes tracing through a program with CodeView (see "Debuggers and CodeView™" on page 173) considerably easier. Normally, you will always use this option during development and skip using it when you want to produce an EXE file you can distribute.

Another useful option, one you would normally use without a filename, is "/?"– the help command. ML, if it encounters this option, will display a list of all the options ML.EXE accepts. Most of these options you will rarely, if ever, use. Consult the MASM documentation for more details on MASM command-line options.

13. By the way, when you create a program using SHELL.ASM it's always a good idea to delete comments like "Insert your global data here." These comments are for the benefit of people reading the SHELL.ASM file, not for people reading your programs. Such comments look really goofy in an actual program.

Typing a command of the form "ML /Zi mypgm.asm" produces two new files (assuming there were no errors): mypgm.obj and mypgm.exe. The OBJ (object code file) is an intermediate file the assembler and *linker* use. Most of the time you can delete this if you program consists of a single source file. The mypgm.exe file is the executable version of the program. You can run this program directly from DOS or run it through the CodeView debugger (often the best choice).

4.9.5 Debuggers and CodeView™

The SIMx86 program is an example of a very simple debugging program. It should come as no surprise that there are several debugger programs available for the 80x86 as well. In this chapter you will learn the basic operation of the CodeView debugger. CodeView is a professional product with many different options and features. This short chapter cannot begin to describe all the possible ways to use the CodeView debugger. However, you will learn how to use some of the more common CodeView commands and debugging techniques.

One major drawback to describing a system like CodeView is that Microsoft constantly updates the CodeView product. These updates create subtle changes in the appearance of several screen images and the operation of various commands. It's quite possible that you're using an older version of CodeView than the one described in this chapter, or this chapter describes an older version of CodeView than the one you're using (This Chapter uses CodeView v4.0). Well, don't let this concern you. The basic principles are the same and you should have no problem adjusting for version differences.

Note: this chapter assumes you are running CodeView from MS-DOS. If you are using a Windows version, the screens will look slightly different.

4.9.5.1 A Quick Look at CodeView

To run CodeView, simply type the following command at the DOS command line prompt:

```
c:> CV program.exe
```

Program.exe represents the name of the program you wish to debug (the ".exe" suffix is optional). CodeView requires that you specify a ".EXE" or ".COM" program name. If you do not supply an executable filename, CodeView will ask you to pick a file when you run it.

CodeView requires an executable program name as the command line parameter. Since you probably haven't written an executable assembly language program yet, you haven't got a program to supply to CodeView. To alleviate this problem, use the SHELL.EXE program found in the Chapter Four subdirectory. To run CodeView using SHELL.EXE just use the command "CV SHELL.EXE". This will bring up a screen which looks something like that in Figure 4.20.

There are four sections to the screen in Figure 4.20: the *menu bar* on the first line, the *source1* window, the *command* window, and the help / status line. Note that CodeView has many windows other than the two above. CodeView remembers which windows were open the last time it was run, so it might come up displaying different windows than those above. At first, the Command window is the active window. However, you can easily switch between windows by pressing the F6 key on the keyboard.

The windows are totally configurable. The Windows menu lets you select which windows appear on the screen. As with most Microsoft windowing products, you select items on the menu bar by holding down the alt key and pressing the first letter of the menu you wish to open. For example, pressing alt-W opens up the Windows menu as shown in Figure 4.21.

Figure 4.20 CodeView Debugger: An Initial Window

Figure 4.21 CodeView Window Menu (alt-W)

4.9.5.2 The Source Window

The Source1 and Source2 items let you open additional source windows. This lets you view, simultaneously, several different sections of the current program you're debugging. Source windows are useful for source level debugging.

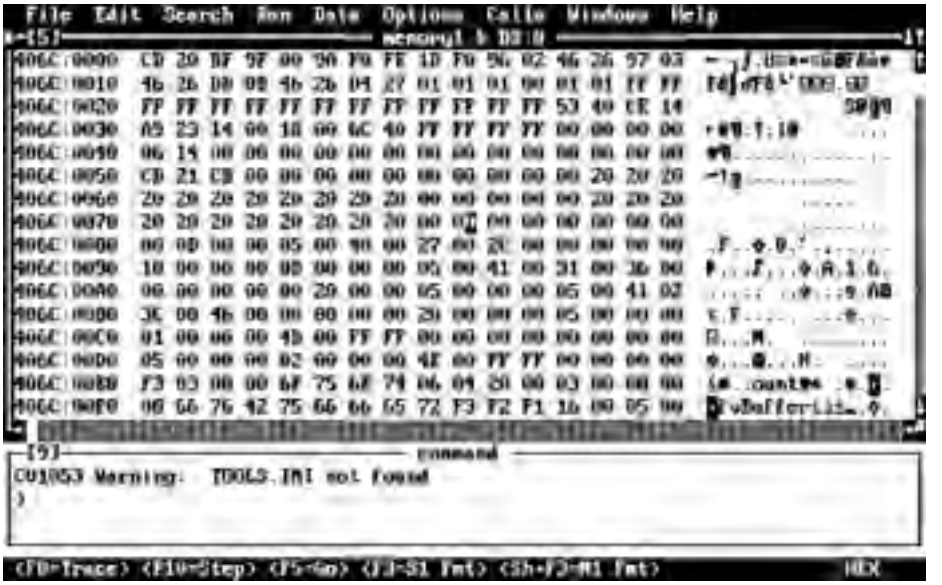


Figure 4.22 A Memory Display

4.9.5.3 The Memory Window

The Memory item lets you open a memory window. The memory windows lets you display and modify values in memory. By default, this window displays the variables in your data segment, though you can easily display any values in memory by typing their address.

Figure 4.22 is an example of a memory display.

The values on the left side of the screen are the segmented memory addresses. The columns of hexadecimal values in the middle of the screen represent the values for 16 bytes starting at the specified address. Finally, the characters on the right hand side of the screen represent the ASCII characters for each of the 16 bytes at the specified addresses. Note that CodeView displays a period for those byte values that are not printable ASCII characters.

When you first bring up the memory window, it typically begins displaying data at offset zero in your data segment. There are a coup.le of ways to display different memory locations. First, you can use the PgUp and PgDn keys to scroll through memory[14]. Another option is to move the cursor over a segment or offset portion of an address and type in a new value. As you type each digit, CodeView automatically displays the data at the new address.

If you want to modify values in memory, simply move the cursor over the top of the desired byte's value and type a new hexadecimal value. CodeView automatically updates the corresponding byte in memory.

CodeView lets you open multiple Memory windows at one time. Each time you select Memory from the View memory, CodeView will open up another Memory window. With multiple memory windows open you can compare the values at several non-contiguous memory locations on the screen at one time. Remember, if you want to switch between the memory windows, press the F6 key.

Pressing Shift-F3 toggles the data display mode between displaying hexadecimal bytes, ASCII characters, words, double words, integers (signed), floating point values, and

14. Mouse users can also move the thumb control on the scroll bar to achieve this same result.

Figure 4.23 The Register Window

other data types. This is useful when you need to view memory using different data types. You only have the option of displaying the contents of the entire window as a single data type; however, you can open multiple memory windows and display a different data type in each one.

4.9.5.4 The Register Window

The Register item in the Windows menu displays or hides the 80x86 registers window. This windows displays the current 80x86 register values (see Figure 4.23).

To change the value of a register, activate the register window (using F6, if it is not already selected) and move the cursor over the value you wish to change. Type a new value over the desired register's existing value. Note that FL stands for *flags*. You can change the values of the flags in the flags register by entering a new value after the FL= entry. Another way to change the flags is to move the cursor over one of the flag entries at the bottom of the register window and press an alphabetic key (e.g., "A") on the keyboard. This will toggle the specified flag. The flag values are (0/1): overflow=(OV/NV), direction=(DN/UP), interrupt=(DI/EI), sign=(PL/NG), zero=(NZ/ZR), auxiliary carry=(NA/AC), parity=(PO/PE), carry=(NC/CY).

Note that pressing the F2 key toggles the display of the registers window. This feature is quite useful when debugging programs. The registers window eats up about 20% of the display and tends to obscure other windows. However, you can quickly recall the registers window, or make it disappear, by simply pressing F2.

4.9.5.5 The Command Window

The **Command** window lets you type textual commands into CodeView. Although almost every command available in the command window is available elsewhere, many operations are easier done in the command window. Furthermore, you can generally execute a sequence of completely different commands in the command window faster than switching between the various other windows in CodeView. The operation of the command window will be the subject of the next section in this chapter.

4.9.5.6 The Output Menu Item

Selecting **View Output** from the **Windows** menu (or pressing the F4 key) toggles the display between the CodeView display and the current program output. While your program is actually running, CodeView normally displays the program's output. Once the program turns control over to CodeView, however, the debugging windows appear obscuring your output. If you need to take a quick peek at the program's output while in CodeView, the F4 key will do the job.

4.9.5.7 The CodeView Command Window

CodeView is actually two debuggers in one. On the one hand, it is a modern window-based debugging system with a nice mouse-based user interface. On the other hand, it can behave like a traditional command-line based debugger. The command window provides the key to this split personality. If you activate the command window, you can enter debugger commands from the keyboard. The following are some of the more common CodeView commands you will use:

A *address*	Assemble
BC *bp_number*	Breakpoint Clear
BD *bp_number*	Breakpoint Disable
BE *bp_number*	Breakpoint Enable
BL	Breakpoint List
BP *address*	Breakpoint Set
D *range*	Dump Memory
E	Animate execution
E*x Address*	Enter Commands (x= " ", b, w, d, etc.)
G {*address*}	Go (address is optional)
H *command*	Help
I *port*	Input data from I/O port
L	Restart program from beginning
MC *range address*	Compare two blocks of memory
MF *range data_value(s)*	Fill Memory with specified value(s)
MM *range address*	Copy a block of memory
MS *range data_value(s)*	Search memory range for set of values
N $Value_{10}$	Set the default radix
O *port value*	Output value to an output port
P	Program Step
Q	Quit
R	Register
R*xx value*	Set register *xx* to *value*
T	Trace
U *address*	Unassemble statements at *address*

In this chapter we will mainly consider those commands that manipulate memory. Execution commands like the breakpoint , trace, and go commands appear in a later chapter. Of course, it wouldn't hurt for you to learn some of the other commands, you may find some of them to be useful.

4.9.5.7.1 The Radix Command (N)

The first command window command you must learn is the RADIX (base selection) command. By default, CodeView works in decimal (base 10). This is very inconvenient for assembly language programmers so you should always execute the radix command upon entering CodeView and set the base to hexadecimal. To do this, use the command

```
N 16
```

Figure 4.24 The Assemble Command

4.9.5.7.2 The Assemble Command

The CodeView command window **Assemble** command works in a fashion not unlike the SIM886 assemble command. The command uses the syntax:

```
A address
```

Address is the starting address of the machine instructions. This is either a full segmented address (*ssss:oooo*, *ssss* is the segment, *oooo* is the offset) or a simple offset value of the form *oooo*. If you supply only an offset, CodeView uses CS' current value as the segment address.

After you press Enter, CodeView will prompt you to enter a sequence of machine instructions. Pressing Enter by itself terminates the entry of assembly language instructions. Figure 4.24 is an example of this command in action.

The Assemble command is one of the few commands available *only* in the command window. Apparently, Microsoft does not expect programmers to enter assembly language code into memory using CodeView. This is not an unreasonable assumption since CodeView is a a high level language source level debugger.

In general, the CodeView Assemble command is useful for quick *patches* to a program, but it is no substitute for MASM 6.x. Any changes you make to your program with the assemble command will not appear in your source file. It's very easy to correct a bug in CodeView and forget to make the change to your original source file and then wonder why the bug is still in your code.

4.9.5.7.3 The Compare Memory Command

The Memory Compare command will compare the bytes in one block of memory against the bytes in a second block of memory. It will report any differences between the two ranges of bytes. This is useful, for example, to see if a program has initialized two arrays in an identical fashion or to compare two long strings. The compare command takes the following forms:

```
MC start_address end_address second_block_address
```

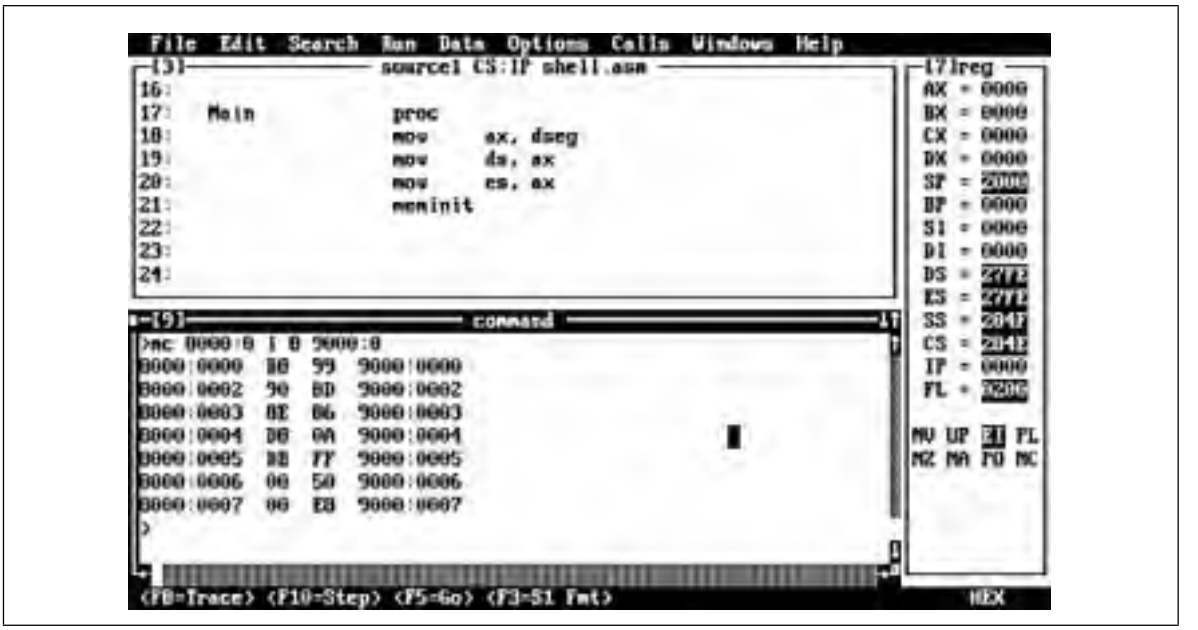


Figure 4.25 The Memory Compare Command

MC *start_address* L *length_of_block second_block_address*

The first form compares the bytes from memory locations *start_address* through *end_address* with the data starting at location *second_block_address.* The second form lets you specify the size of the blocks rather than specify the ending address of the first block. If CodeView detects any differences in the two ranges of bytes, it displays those differences and their addresses. The following are all legal compare commands:

```
MC 8000:0 8000:100 9000:80
MC 8000:100 L 20 9000:0
MC 0 100 200
```

The first command above compares the block of bytes from locations 8000:0 through 8000:100 against a similarly sized block starting at address 9000:80 (i.e., 9000:80..180).

The second command above demonstrates the use of the “L” option which specifies a length rather than an ending address. In this example, CodeView will compare the values in the range 8000:0..8000:1F (20h / 32 bytes) against the data starting at address 9000:0.

The third example above shows that you needn’t supply a full segmented address for the *starting_address* and *second_block_address* values. By default, CodeView uses the data segment (DS:) if you do not supply a segment portion of the address. Note, however, that if you supply a starting and ending address, they must both have the same segment value; you must supply the same segment address to both or you must let both addresses default to DS’ value.

If the two blocks are equal, CodeView immediately prompts you for another command without printing anything to the command window. If there are differences between the two blocks of bytes, however, CodeView lists those differences (and their addresses) in the command window.

In the example in Figure 4.25, memory locations 8000:0 through 8000:200 were first initialized to zero. Then locations 8000:10 through 8000:1E were set to 1, 2, 3, ..., 0Fh. Finally, the Memory Compare command compared the bytes in the range 8000:0…8000:FF with the block of bytes starting at address 8000:100. Since locations 8000:10…8000:1E were different from the bytes at locations 8000:110…8000:11E, CodeView printed their addresses and differences.

Figure 4.26 The Memory Dump Command

4.9.5.7.4 The Dump Memory Command

The Dump command lets you display the values of selected memory cells. The Memory window in CodeView also lets you view (and modify) memory. However, the Dump command is sometimes more convenient, especially when looking at small blocks of memory.

The Dump command takes several forms, depending on the type of data you want to display on the screen. This command typically takes one of the forms:

```
D starting_address ending_address
D starting_address L length
```

By default, the dump command displays 16 hexadecimal and ASCII byte values per line (just like the Memory window).

There are several additional forms of the Dump command that let you specify the display format for the data. However, the exact format seems to change with every version of CodeView. For example, in CodeView 4.10, you would use commands like the following:

```
DA address_range        Dump ASCII characters
DB address_range        Dump hex bytes/ASCII (default)
DI address_range        Dump integer words
DIU address_range       Dump unsigned integer words
DIX address_range       Dump 16-bit values in hex
DL address_range        Dump 32-bit integers
DLU address_range       Dump 32-bit unsigned integers
DLX address_range       Dump 32-bit values in hex
DR address_range        Dump 32-bit real values
DRL address_range       Dump 64-bit real values
DRT address_range       Dump 80-bit real values
```

You should probably check the help associated with your version of CodeView to verify the exact format of the memory dump commands. Note that some versions of CodeView allow you to use MDxx for the memory dump command.

Once you execute one of the above commands, the “D” command name displays the data in the new format. The “DB” command reverts back to byte/ASCII display. Figure 4.26 provides an example of these commands.

If you enter a dump command without an address, CodeView will display the data immediately following the last dump command. This is sometimes useful when viewing memory.

4.9.5.7.5 The Enter Command

The CodeView Memory windows lets you easily display and modify the contents of memory. From the command window it takes two different commands to accomplish these tasks: Dump to display memory data and Enter to modify memory data. For most memory modification tasks, you'll find the memory windows easier to use. However, the CodeView Enter command handles a few tasks easier than the Memory window.

Like the Dump command, the Enter command lets you enter data in several different formats. The commands to accomplish this are

```
EA-     Enter data in ASCII format
EB-     Enter byte data in hexadecimal format
ED-     Enter double word data in hexadecimal format
EI-     Enter 16-bit integer data in (signed) decimal format
EIU-    Enter 16-bit integer data in (unsigned) decimal format.
EIX-    Enter 16-bit integer data in hexadecimal format.
EL-     Enter 32-bit integer data in (signed) decimal format
ELU-    Enter 32-bit integer data in (unsigned) decimal format.
ELX-    Enter 32-bit integer data in hexadecimal format.
ER-     Enter 32-bit floating point data
ERL-    Enter 64-bit floating pont data
ERT-    Enter 80-bit floating point data
```

Like the Dump command, the syntax for this command changes regularly with different versions of CodeView. Be sure to use CodeView's help facility if these commands don't seem to work. *MExx* is a synonym for *Exx* in CodeView.

Enter commands take two possible forms:

```
Ex starting_address
Ex starting_address list_of_values
```

The first form above is the *interactive Enter command.* Upon pressing the key, Codeview will display the starting address and the data at that address, then prompt you to enter a new value for that location. Type the new value followed by a space and CodeView will prompt you for the value for the next location; typing a space by itself skips over the current location; typing the enter key or a value terminated with the enter key terminates the interactive Enter mode. Note that the EA command does not let you enter ASCII values in the interactive mode. It behaves exactly like the EB command during data entry.

The second form of the Enter command lets you enter a sequence of values into memory a single entry. With this form of the Enter command, you simply follow the starting address with the list of values you want to store at that address. CodeView will automatically store each value into successive memory locations beginning at the starting address. You can enter ASCII data using this form of Enter by enclosing the characters in quotes. Figure 4.27 demonstrates the use of the Enter command.

There are a couple of points concerning the Enter command of which you should be aware. First of all, you cannot use "E" as a command by itself . Unlike the Dump command, this does not mean "begin entering data after the last address." Instead, this is a totally separate command (Animate). The other thing to note is that the current display mode (ASCII, byte, word, double word, etc.) and the current entry mode are not independent. Changing the default display mode to word also changes the entry mode to word, and vice versa.

Figure 4.27 The Enter Command

4.9.5.7.6 The Fill Memory Command

The Enter command and the Memory window let you easily change the value of individual memory locations, or set a range of memory locations to several different values. If you want to clear an array or otherwise initialize a block of memory locations so that they all contain the same values, the Memory Fill command provides a better alternative.

The Memory Fill command uses the following syntax:

```
MF starting_address ending_address values
MF starting_address L block_length values
```

The Memory Fill command fills memory locations *starting_address* through *ending_address* with the byte values specified in the *values* list. The second form above lets you specify the block length rather than the ending address.

The *values* list can be a single value or a list of values. If *values* is a single byte value, then the Memory Fill command initializes all the bytes of the memory block with that value. If *values* is a list of bytes, the Fill command repeats that sequence of bytes over and over again in memory. For example, the following command stores 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5... to the 256 bytes starting at location 8000:0

```
F 8000:0 L 100 1 2 3 4 5
```

Unfortunately, the Fill command works only with byte (or ASCII string) data. However, you can simulate word, doubleword, etc., memory fills breaking up those other values into their component bytes. Don't forget, though, that the L.O. byte always comes first.

4.9.5.7.7 The Move Memory Command

This Command window operation copies data from one block of memory to another. This lets you copy the data from one array to another, move code around in memory, reinitialize a group of variables from a saved memory block, and so on. The syntax for the Memory Move command is as follows:

```
MM starting_address ending_address destination_address
MM starting_address L block_length destination_address
```

If the source and destination blocks overlap, CodeView detects this and handles the memory move operation correctly.

4.9.5.7.8 The Input Command

The Input command lets you read data from one of the 80x86's 65,536 different input ports. The syntax for this command is

```
I port_address
```

where *port_address* is a 16-bit value denoting the I/O port address to read. The input command reads the byte at that port and displays its value.

Note that it is not a wise idea to use this command with an arbitrary address. Certain devices activate some functions whenever you read one of their I/O ports. By reading a port you may cause the device to lose data or otherwise disturb that device.

Note that this command only reads a single byte from the specified port. If you want to read a word or double-word from a given input port you will need to execute two successive Input operations at the desired port address and the next port address.

This command appears to be broken in certain versions of CodeView (e.g., 4.01).

4.9.5.7.9 The Output Command

The Output command is complementary to the Input command. This command lets you output a data value to a port. It uses the syntax:

```
O port_address output_value
```

Output_value is a single byte value that CodeView will write to the output port given by *port_address*.

Note that CodeView also uses the "O" command to set options. If it does not recognize a valid port address as the first operand it will think this is an Option command. If the Output command doesn't seem to be working properly, you've probably switched out of the assembly language mode (CodeView supports BASIC, Pascal, C, and FORTRAN in addition to assembly language) and the port address you're entering isn't a valid numeric value in the new mode. Be sure to use the N 16 command to set the default radix to hexadecimal before using this command!

4.9.5.7.10 The Quit Command

Pressing Q (for Quit) terminates the current debugging session and returns control to MS-DOS. You can also quit CodeView by selecting the Exit item from the File menu.

4.9.5.7.11 The Register Command

The CodeView Register command lets you view and change the values of the registers. To view the current values of the 80x86 registers you would use the following command:

```
R
```

Figure 4.28 The Register Command

This command displays the registers and disassembles the instruction at address CS:IP.

You can also change the value of a specific register using a command of the form:

R*xx*
-or-
R*xx* = *value*

where *xx* represents one of the 80x86's register names: AX, BX, CX, DX, SI, DI, BP, SP, CS, DS, ES, SS, IP, or FL. The first version ("R*xx*") displays the specified register and then prompts you to enter a new value. The second form of this command above immediately sets the specified register to the given value (see Figure 4.28).

4.9.5.7.12 The Unassemble Command

The Command window Unassemble command will disassemble a sequence of instructions at an address you specify, converting the binary machine codes into (barely) readable machine instructions. The basic command uses the following syntax:

U *address*
Note that you must have a source window open for this instruction to operate properly!

In general, the Unassemble command is of little use because the Source window lets you view your program at the source level (rather than at the disassembled machine language level). However, the Unassemble command is great for disassembling BIOS, DOS, TSRs, and other code in memory.

4.9.5.8 CodeView Function Keys

CodeView uses the function keys on the PC's keyboard for often-executed operations. The following table gives a brief description of the use of each function key.

Table 26: Function Key Usage in CodeView

Function Key	Alone	Shift	Ctrl	Alt
F1	Help	Help contents	Next Help	Prev Help
F2	Register Window			
F3	Source Window Mode	Memory Window Mode		
F4	Output Screen		Close Window	
F5	Run			
F6	Switch Window	Prev Window		
F7	Execute to cursor			
F8	Trace	Prev History	Size window	
F9	Breakpoint			
F10	Step instrs, run calls.	Next History	Maximize Window	

The F3 function key deserves special mention. Pressing this key toggles the source mode between *machine language* (actually, disassembled machine language), *mixed*, and *source*. In source mode (assuming you've assembled your code with the proper options) the source window displays your actual source code. In mixed mode, CodeView displays a line of source code followed by the machine code generated for that line of source code. This mode is primarily for high level language users, but it does have some utility for assembly language users as you'll see when you study macros. In *machine mode*, CodeView ignores your source code and simply disassembles the binary opcodes in memory. This mode is useful if you suspect a bug in MASM (they do exist) and you're not sure than MASM is assembling your code properly.

4.9.5.9 Some Comments on CodeView Addresses

The examples given for addresses in the previous sections are a little misleading. You could easily get the impression that you have to enter an address in hexadecimal form, i.e., *ssss:oooo* or *oooo*. Actually, you can specify memory addresses in many different ways. For example, if you have a variable in your assembly language program named *MyVar*, you could use a command like

```
D Myvar
```

to display the value of this variable[15]. You do not need to know the address, nor even the segment of that variable. Another way to specify an address is via the 80x86 register set. For example, if ES:BX points at the block of memory you want to display, you could use the following command to display your data:

```
D ES:BX
```

CodeView will use the current values in the `es` and `bx` registers as the address of the block of memory to display. There is nothing magical about the use of the registers. You can use them just like any other address component. In the example above, `es` held the segment value and `bx` held the offset— very typical for an 80x86 assembly language program.

15. This requires that you assemble your program in a very special way, but we're getting to that.

However, CodeView does not require you to use legal 80x86 combinations. For example, you could dump the bytes at address cx:ax using the dump command

```
D CX:AX
```

The use of 80x86 registers is not limited to specifying source addresses. You can specify destination addresses and even lengths using the registers:

```
D CX:AX L BX ES:DI
```

Of course, you can mix and match the use of registers and numeric addresses in the same command with no problem:

```
D CX:AX L 100 8000:0
```

You can also use complex arithmetic expressions to specify an address in memory. In particular, you can use the addition operator to compute the sum of various components of an address. This works out really neat when you need to simulate 80x86 addressing modes. For example, if you want to see which byte is at address 1000[bx], you could use the command:

```
D BX+1000 L 1
```

To simulate the [BX][SI] addressing mode and look at the word at that address you could use the command:

```
DIX BX+SI L 1
```

The examples presented in this section all use the Dump command, but you can use this technique with any of the CodeView commands. For more information concerning what constitutes valid CodeView address, as well as a full explanation of allowable expression forms, please consult the CodeView on-line help system.

4.9.5.10 A Wrap on CodeView

We're not through discussing CodeView by any means. In particular, we've not discussed the execution, single stepping, and breakpoint commands which are crucial for debugging programs. We will return to these subjects in later chapters. Nonetheless, we've covered a considerable amount of material, certainly enough to deal with most of the experiments in this laboratory exercise. As we need those other commands, this manual will introduce them.

Of course, there are two additional sources of information on CodeView available to you— the section on CodeView in the "Microsoft Macro Assembler Programmer's Guide" and the on-line help available inside CodeView. In particular, the on-line help is quite useful for figuring out how a specific command works inside CodeView.

4.9.6 Laboratory Tasks

The Chapter Four subdirectory on the companion CD-ROM contains a sample file named EX4_1.ASM. Assemble this program using MASM (do not use the /Zi option for the time being). **For your lab report:** include a print-out of the program. Describe what the program does. Run the program and include a print-out of the program's output with your lab report.

Whenever you assemble a program MASM, by default, writes one byte of data to the file for every instruction byte and data variable in the program, even if that data is uninitialized. If you declare large arrays in your program the EXE file ML produces will be quite large as well. Note the size of the EX4_1.EXE program you created above. Now reassemble the program using the following command:

```
ml EX4_1.asm /link /exepack
```

ML passes the "/link /exepack" option on to the linker. The exepack option tells the linker to pack the EXE file by removing redundant information (in particular, the unini-

tialized data). This often makes the EXE file much smaller. **For your lab report:** after assembling the file using the command above, note the size of the resulting EXE file. Compare the two sizes and comment on their difference in your lab report.

Note that the EXEPACK option only saves disk space. It does not make the program use any less memory while it is running. Furthermore, you cannot load programs you've packed with the EXEPACK option into the CodeView debugger. Therefore, you should not use the EXEPACK option during program development and testing. You should only use this option once you've eliminated all the bugs from the program and further development ceases.

Using your editor of choice, edit the x86.asm file. Read the comments at the beginning of the program that explain how to write x86 programs that assemble and run on the 80x86 CPU. **For your lab report:** describe the restrictions on the x86 programs you can write.

The EX4_2.ASM source file is a copy of the x86.ASM file with a few additional comments in the main program describing a set of procedures you should follow. Load this file into your text editor of choice and read the instructions in the main program. Follow them to produce a program. Assemble this program using ML and execute the resulting EX4_2.EXE program file. **For your lab report:** include a print-out of your resulting program. Include a print-out of the program's output when you run it.

Trying loading EX4_2.EXE into CodeView using the following DOS Window command:

```
cv EX4_2
```

When CodeView runs you will notice that it prints a message in the command window complaining that there is "no CodeView information for EX4_2.EXE." Look at the code in the source window. Try and find the instructions you place in the main program. **For your lab report:** contrast the program listing appearing in the CodeView source window with that produced on the Emulator screen of the SIMx86 program.

Now reassemble the EX4_2.asm file and load it into CodeView using the following DOS commands:

```
ml /Zi EX4_2.asm
cv EX4_2
```

For your lab report: describe the difference in the CodeView source window when using the /Zi ML option compared to the CodeView source window without this option.

4.10 Programming Projects

Note: You are to write these programs in 80x86 assembly language code using a copy of the X86.ASM file as the starting point for your programs. The 80x86 instruction set is almost a superset of the x86 instruction set. Therefore, you can use most of the instructions you learned in the last chapter. Read the comments at the beginning of the x86.ASM file for more details. **Note in particular that you cannot use the label "C" in your program because "C" is a reserved word in MASM.** Include a specification document, a test plan, a program listing, and sample output with your program submissions.

1) The following projects are modifications of the programming assignments in the previous chapter. Convert those x86 programs to their 80x86 counterparts.

1a. The x86 instruction set does not include a multiply instruction. Write a short program that reads two values from the user and displays their product (hint: remember that multiplication is just repeated addition).

1b. Write a program that reads three values from the user: an address it puts into BX, a count it puts into CX, and a value it puts in AX. It should write CX copies of AX to successive words in memory starting at address BX (in the data segment).

1c. Write the generic logic function for the x86 processor (see Chapter Two). Hint: add ax, ax does a shift left on the value in ax. You can test to see if the high order bit is set by checking to see if ax is greater than 8000h.

1d. Write a program that scans an array of words starting at address 1000h and memory, of the length specified by the value in cx, and locates the maximum value in that array. Display the value after scanning the array.

1e. Write a program that computes the two's complement of an array of values starting at location 1000h. CX should contain the number of values in the array. Assume each array element is a two-byte integer.

1f. Write a simple program that *sorts* the words in memory locations 1000..10FF in ascending order. You can use a simple *insertion sort* algorithm. The Pascal code for such a sort is

```
for i := 0 to n-1 do
       for j := i+1 to n do
                 if (memory[i] > memory[j]) then
                 begin
                           temp := memory[i];
                           memory[i] := memory[j];
                           memory[j] := temp;
                 end;
```

For the following projects, feel free to use any additional 80x86 addressing modes that might make the project easier to write.

2) Write a program that stores the values 0, 1, 2, 3, ..., into successive words in the data segment starting at offset 3000h and ending at offset 3FFEh (the last value written will be 7FFh). Then store the value 3000h to location 1000h. Next, write a code segment that sums the 512 words starting at the address found in location 1000h. This portion of the program cannot assume that 1000h contains 3000h. Print the sum and then quit.

4.11 Summary

This chapter presents an 80x86-centric view of memory organization and data structures. This certainly isn't a complete course on data structures, indeed this topic appears again later in Volume Two. This chapter discussed the primitive and simple composite data types and how to declare and use them in your program. Lots of additional information on the declaration and use of simple data types appears in "MASM: Directives & Pseudo-Opcodes" on page 355.

The 8088, 8086, 80188, 80186, and 80286 all share a common set of registers which typical programs use. This register set includes the general purpose registers: ax, bx, cx, dx, si, di, bp, and sp; the segment registers: cs, ds, es, and ss; and the special purpose registers ip and flags. These registers are 16 bits wide. These processors also have eight 8 bit registers: al, ah, bl, bh, cl, ch, dl, and dh which overlap the ax, bx, cx, and dx registers. See:

- "8086 General Purpose Registers" on page 146
- "8086 Segment Registers" on page 147
- "8086 Special Purpose Registers" on page 148

In addition, the 80286 supports several special purpose memory management registers which are useful in operating systems and other system level programs. See:

- "80286 Registers" on page 148

The 80386 and later processors extend the general purpose and special purpose register sets to 32 bits. These processors also add two additional segment registers you can use in your application programs. In addition to these improvements, which any program can take advantage of, the 80386/486 processors also have several additional system level registers for memory management, debugging, and processor testing. See:

- "80386/80486 Registers" on page 149

The Intel 80x86 family uses a powerful memory addressing scheme known as *segmented addressing* that provides simulated two dimensional addressing. This lets you group logically related blocks of data into segments. The exact format of these segments depends on whether the CPU is operating in *real mode* or *protected mode*. Most DOS programs operate in real mode. When working in real mode, it is very easy to convert a *logical* (segmented) address to a linear *physical* address. However, in protected mode this conversion is considerably more difficult. See:

- "Segments on the 80x86" on page 151

Because of the way segmented addresses map to physical addresses in real mode, it is quite possible to have two different segmented addresses that refer to the same memory location. One solution to this problem is to use normalized addresses. If two normalized addresses do not have the same bit patterns, they point at different addresses. Normalized pointers are useful when comparing pointers in real mode. See:

- "Normalized Addresses on the 80x86" on page 154

With the exception of two instructions, the 80x86 doesn't actually work with full 32 bit segmented addresses. Instead, it uses *segment registers* to hold default segment values. This allowed Intel's designers to build a much smaller instruction set since addresses are only 16 bits long (offset portion only) rather than 32 bits long. The 80286 and prior processors provide four segment registers: `cs`, `ds`, `es`, and `ss`; the 80386 and later provide six segment registers: `cs`, `ds`, `es`, `fs`, `gs`, and `ss`. See:

- "Segment Registers on the 80x86" on page 155

The 80x86 family provides many different ways to access variables, constants, and other data items. The name for a mechanism by which you access a memory location is *addressing mode*. The 8088, 8086, and 80286 processors provide a large set of memory addressing modes. See:

- "The 80x86 Addressing Modes" on page 155
- "8086 Register Addressing Modes" on page 156
- "8086 Memory Addressing Modes" on page 156

The 80386 and later processors provide an expanded set of register and memory addressing modes. See:

- "80386 Register Addressing Modes" on page 163
- "80386 Memory Addressing Modes" on page 163

The most common 80x86 instruction is the `mov` instruction. This instruction supports most of the addressing modes available on the 80x86 processor family. Therefore, the `mov` instruction is a good instruction to look at when studying the encoding and operation of 80x86 instructions. See:

- "The 80x86 MOV Instruction" on page 166

The `mov` instruction takes several generic forms, allowing you to move data between a register and some other location. The possible source/destination locations include: (1) other registers, (2) memory locations (using a general memory addressing mode), (3) constants (using the immediate addressing mode), and (4) segment registers.

The `mov` instruction lets you transfer data between two locations (although you cannot move data between two memory locations see the discussion of the mod-reg-r/m byte).

4.12 Questions

1) Although the 80x86 processors always use segmented addresses, the instruction encodings for instructions like "`mov AX, I`" only have a 16 bit offset encoded into the opcode. Explain.

2) Segmented addressing is best described as a *two dimensional addressing scheme*. Explain.

3) Convert the following logical addresses to physical addresses. Assume all values are hexadecimal and real mode operation on the 80x86:

   a) 1000:1000 b) 1234:5678 c) 0:1000 d) 100:9000 e) FF00:1000

   f) 800:8000 g) 8000:800 h) 234:9843 i) 1111:FFFF j) FFFF:10

4) Provide *normalized* forms of the logical addresses above.

5) List all the 8086 memory addressing modes.

6) List all the 80386 (and later) addressing mode that are not available on the 8086 (use generic forms like disp[reg], do no enumerate all possible combinations).

7) Besides memory addressing modes, what are the other two major addressing modes on the 8086?

8) Describe a common use for each of the following addressing modes:

   a) Register b) Displacement only c) Immediate

   d) Register Indirect e) Indexed f) Based indexed

   g) Based indexed plus displacement h) Scaled indexed

9) Given the bit pattern for the generic MOV instruction (see "The 80x86 MOV Instruction" on page 166) explain why the 80x86 does not support a memory to memory move operation.

10) Which of the following MOV instructions are *not* handled by the generic MOV instruction opcode? Explain.

    a) mov ax, bx b) mov ax, 1234 c) mov ax, I

    d) mov ax, [bx] e) mov ax, ds f) mov [bx], 2

11) Assume the variable "I" is at offset 20h in the data segment. Provide the binary encodings for the above instructions.

12) What determines if the R/M field specifies a register or a memory operand?

13) What field in the REG-MOD-R/M byte determines the size of the displacement following an instruction? What displacement sizes does the 8086 support?

14) Why doesn't the displacement only addressing mode support multiple displacement sizes?

15) Why would you *not* want to interchange the two instructions "mov ax, [bx]" and "mov ax,[ebx]"?

16) Certain 80x86 instructions take several forms. For example, there are two different versions of the MOV instruction that load a register with an immediate value. Explain why the designers incorporated this redundancy into the instruction set.

17) Why isn't there a true [bp] addressing mode?

18) List all of the 80x86 eight bit registers.

19) List all the 80x86 general purpose 16 bit registers.

20) List all the 80x86 segment registers (those available on all processors).

21) Describe the "special purposes" of each of the general purpose registers.

22) List all the 80386/486/586 32 bit general purpose registers.

23) What is the relationship between the 8, 16, and 32 bit general purpose registers on the 80386?

24) What values appear in the 8086 flags register? The 80286 flags register?

25) Which flags are the condition codes?

26) Which extra segment registers appear on the 80386 but not on earlier processors?

Variables and Data Structures Chapter Five

Chapter One discussed the basic format for data in memory. Chapter Three covered how a computer system physically organizes that data. This chapter finishes this discussion by connecting the concept of *data representation* to its actual physical representation. As the title implies, this chapter concerns itself with two main topics: variables and data structures. This chapter does not assume that you've had a formal course in data structures, though such experience would be useful.

5.0 Chapter Overview

This chapter discusses how to declare and access scalar variables, integers, reals, data types, pointers, arrays, and structures. You must master these subjects before going on to the next chapter. Declaring and accessing arrays, in particular, seems to present a multitude of problems to beginning assembly language programmers. However, the rest of this text depends on your understanding of these data structures and their memory representation. Do not try to skim over this material with the expectation that you will pick it up as you need it later. You will need it right away and trying to learn this material along with later material will only confuse you more.

5.1 Some Additional Instructions: LEA, LES, ADD, and MUL

The purpose of this chapter is not to present the 80x86 instruction set. However, there are four additional instructions (above and beyond mov) that will prove handy in the discussion throughout the rest of this chapter. These are the *load effective address* (lea), *load es and general purpose register* (les), *addition* (add), and *multiply* (mul). These instructions, along with the mov instruction, provide all the necessary power to access the different data types this chapter discusses.

The lea instruction takes the form:

```
lea     reg16, memory
```

reg16 is a 16 bit general purpose register. *Memory* is a memory location represented by a mod/reg/rm byte[1] (except it must be a memory location, it cannot be a register).

This instruction loads the 16 bit register with the offset of the location specified by the memory operand. lea ax,1000h[bx][si], for example, would load ax with the address of the memory location pointed at by 1000h[bx][si]. This, of course, is the value 1000h+bx+si. Lea is also quite useful for obtaining the address of a variable. If you have a variable I somewhere in memory, lea bx,I will load the bx register with the address (offset) of I.

The les instruction takes the form

```
les     reg16, memory32
```

This instruction loads the es register and one of the 16 bit general purpose registers from the specified memory address. Note that any memory address you can specify with a mod/reg/rm byte is legal but like the lea instruction it must be a memory location, not a register.

The les instruction loads the specified general purpose register from the word at the given address, it loads the es register from the following word in memory. This instruction, and it's companion lds (which loads ds) are the only instructions on pre-80386 machines that manipulate 32 bits at a time.

1. Or by the mod/reg/rm -- sib addressing mode bytes on the 80386.

The add instruction, like it's x86 counterpart, adds two values on the 80x86. This instruction takes several forms. There are five forms that concern us here. They are

```
add     reg, reg
add     reg, memory
add     memory, reg
add     reg, constant
add     memory, constant
```

All these instructions add the second operand to the first leaving the sum in the first operand. For example, add bx,5 computes bx := bx + 5.

The last instruction to look at is the mul (multiply) instruction. This instruction has only a single operand and takes the form:

```
mul     reg/memory
```

There are many important details concerning mul that this chapter ignores. For the sake of the discussion that follows, assume that the register or memory location is a 16 bit register or memory location. In such a case this instruction computes dx:ax :=ax*reg/mem[2]. Note that there is no immediate mode for this instruction.

5.2 Declaring Variables in an Assembly Language Program

Although you've probably surmised that memory locations and variables are somewhat related, this chapter hasn't gone out of its way to draw strong parallels between the two. Well, it's time to rectify that situation. Consider the following short (and useless) Pascal program:

```
program useless(input,output);
var i,j:integer;
begin
                i := 10;
                write('Enter a value for j:');
                readln(j);
                i := i*j + j*j;
                writeln('The result is ',i);
end.
```

When the computer executes the statement i:=10;[3] it makes a copy of the value 10 and somehow remembers this value for use later on. To accomplish this, the compiler sets aside a memory location specifically for the exclusive use of the variable i. Assuming the compiler arbitrarily assigned location DS:10h for this purpose it could use the instruction mov ds:[10h],10 to accomplish this[4]. If i is a 16 bit word, the compiler would probably assign the variable j to the word starting at location 12h or 0Eh. Assuming it's location 12h, the second assignment statement in the program might wind up looking like the following:

```
mov     ax, ds:[10h]            ;Fetch value of I
mul     ds:[12h]                ;Multiply by J
mov     ds:[10h], ax            ;Save in I (ignore overflow)
mov     ax, ds:[12h]            ;Fetch J
mul     ds:[12h]                ;Compute J*J
add     ds:[10h], ax            ;Add I*J + J*J, store into I
```

2. Any time you multiply two 16 bit values you could get a 32 bit result. The 80x86 places this 32 bit result in dx:ax with the H.O. word in dx and the L.O. word in ax.

3. Actually, the computer executes the *machine code* emitted by the Pascal compiler for this statement; but you need not worry about such details here.

4. But don't try this at home, folks! There is one minor syntactical detail missing from this instruction. MASM will complain bitterly if you attempt to assemble this particular instruction.

Although there are a few details missing from this code, it is fairly straightforward and you can easily see what is going on in this program.

Now imagine a 5,000 line program like this one using variables like ds:[10h], ds:[12h], ds:[14h**],** etc. Would you want to locate the statement where you accidentally stored the result of a computation into j rather than i? Indeed, why should you even care that the variable i is at location 10h and j is at location 12h? Why shouldn't you be able to use names like i and j rather than worrying about these numerical addresses? It seems reasonable to rewrite the code above as:

```
mov     ax, i
mul     j
mov     i, ax
mov     ax, j
mul     j
add     i, ax
```

Of course you can do this in assembly language! Indeed, one of the primary jobs of an assembler like MASM is to let you use symbolic names for memory locations. Furthermore, the assembler will even assign locations to the names automatically for you. You needn't concern yourself with the fact that variable i is really the word at memory location DS:10h unless you're curious.

It should come as no surprise that ds will point to the dseg segment in the SHELL.ASM file. Indeed, setting up ds so that it points at dseg is one of the first things that happens in the SHELL.ASM main program. Therefore, all you've got to do is tell the assembler to reserve some storage for your variables in dseg and attach the offset of said variables with the names of those variables. This is a very simple process and is the subject of the next several sections.

5.3 Declaring and Accessing Scalar Variables

Scalar variables hold single values. The variables i and j in the preceding section are examples of scalar variables. Examples of data structures that are not scalars include arrays, records, sets, and lists. These latter data types are made up from scalar values. They are the *composite types*. You'll see the composite types a little later; first you need to learn to deal with the scalar types.

To declare a variable in dseg, you would use a statement something like the following:

```
ByteVar         byte    ?
```

ByteVar is a *label*. It should begin at column one on the line somewhere in the dseg segment (that is, between the dseg segment and dseg ends statements). You'll find out all about labels in a few chapters, for now you can assume that most legal Pascal / C / Ada identifiers are also valid assembly language labels.

If you need more than one variable in your program, just place additional lines in the dseg segment declaring those variables. MASM will automatically allocate a unique storage location for the variable (it wouldn't be too good to have i and j located at the same address now, would it?). After declaring said variable, MASM will allow you to refer to that variable *by name* rather than by location in your program. For example, after inserting the above statement into the data segment (dseg), you could use instructions like mov ByteVar,al in your program.

The first variable you place in the data segment gets allocated storage at location DS:0. The next variable in memory gets allocated storage just beyond the previous variable. For example, if the variable at location zero was a byte variable, the next variable gets allocated storage at DS:1. However, if the first variable was a word, the second variable gets allocated storage at location DS:2. MASM is always careful to allocate variables in such a manner that they do not overlap. Consider the following dseg definition:

```
dseg            segment  para public 'data'
bytevar         byte     ?                      ;byte allocates bytes
wordvar         word     ?                      ;word allocates words
dwordvar        dword    ?                      ;dword allocs dbl words
byte2           byte     ?
word2           word     ?
dseg            ends
```

MASM allocates storage for *bytevar* at location DS:0. Because *bytevar* is one byte long, the next available memory location is going to be DS:1. MASM, therefore, allocates storage for *wordvar* at location DS:1. Since words require two bytes, the next available memory location after *wordvar* is DS:3 which is where MASM allocates storage for *dwordvar. Dwordvar* is four bytes long, so MASM allocates storage for *byte2* starting at location DS:7. Likewise, MASM allocates storage for *word2* at location DS:8. Were you to stick another variable after *word2*, MASM would allocate storage for it at location DS:0A.

Whenever you refer to one of the names above, MASM automatically substitutes the appropriate offset. For example, MASM would translate the mov ax,wordvar instruction into mov ax,ds:[1]. So now you can use symbolic names for your variables and completely ignore the fact that these variables are really memory locations with corresponding offsets into the data segment.

5.3.1 Declaring and using BYTE Variables

So what are byte variables good for, anyway? Well you can certainly represent any data type that has less than 256 different values with a single byte. This includes some very important and often-used data types including the character data type, boolean data type, most enumerated data types, and small integer data types (signed and unsigned), just to name a few.

Characters on a typical IBM compatible system use the eight bit ASCII / IBM character set (see "A: ASCII / IBM Character Set" on page 1345). The 80x86 provides a rich set of instructions for manipulating character data. It's not surprising to find that most byte variables in a typical program hold character data.

The boolean data type represents only two values: true or false. Therefore, it only takes a single bit to represent a boolean value. However, the 80x86 really wants to work with data at least eight bits wide. It actually takes extra code to manipulate a single bit rather than a whole byte. Therefore, you should use a whole byte to represent a boolean value. Most programmers use the value zero to represent false and anything else (typically one) to represent true. The 80x86's zero flag makes testing for zero / not zero very easy. Note that this choice of zero or non-zero is mainly for convenience. You could use *any* two different values (or two different sets of values) to represent true and false.

Most high level languages that support enumerated data types convert them (internally) to unsigned integers. The first item in the list is generally item zero, the second item in the list is item one, the third is item two, etc. For example, consider the following Pascal enumerated data type:

```
colors = (red, blue, green, purple, orange, yellow, white, black);
```

Most Pascal compilers will assign the value zero to red, one to blue, two to green, etc.

Later, you will see how to actually create your own enumerated data types in assembly language. All you need to learn now is how to allocate storage for a variable that holds an enumerated value. Since it's unlikely there will be more than 256 items enumerated by the data type, you can use a simple byte variable to hold the value. If you have a variable, say *color* of type *colors*, using the instruction mov color,2 is the same thing as saying color:=green in Pascal. (Later, you'll even learn how to use more meaningful statements like mov color,green to assign the color green to the color variable).

Of course, if you have a small unsigned integer value (0…255) or small signed integer (-128…127) a single byte variable is the best way to go in most cases. Note that most pro-

grammers treat all data types except small signed integers as unsigned values. That is, characters, booleans, enumerated types, and unsigned integers are all usually unsigned values. In some very special cases you might want to treat a character as a signed value, but most of the time even characters are unsigned values.

There are three main statements for declaring byte variables in a program. They are

```
identifier      db      ?
identifier      byte    ?
and
identifier      sbyte   ?
```

identifier represents the name of your byte variable. "db" is an older term that predates MASM 6.x. You will see this directive used quite a bit by other programmers (especially those who are not using MASM 6.x or later) but Microsoft considers it to be an obsolete term; you should always use the byte and sbyte declarations instead.

The byte declaration declares unsigned byte variables. You should use this declaration for all byte variables *except* small signed integers. For signed integer values, use the sbyte (signed byte) directive.

Once you declare some byte variables with these statements, you may reference those variables within your program by their names:

```
i               db      ?
j               byte    ?
k               sbyte   ?
                  .
                  .
                  .
                mov     i, 0
                mov     j, 245
                mov     k, -5
                mov     al, i
                mov     j, al
                etc.
```

Although MASM 6.x performs a small amount of type checking, you should not get the idea that assembly language is a strongly typed language. In fact, MASM 6.x will only check the values you're moving around to verify that they will *fit* in the target location. All of the following are legal in MASM 6.x:

```
                mov     k, 255
                mov     j, -5
                mov     i, -127
```

Since all of these variables are byte-sized variables, and all the associated constants will fit into eight bits, MASM happily allows each of these statements. Yet if you look at them, they are logically incorrect. What does it mean to move -5 into an unsigned byte variable? Since signed byte values must be in the range -128…127, what happens when you store the value 255 into a signed byte variable? Well, MASM simply converts these values to their eight bit equivalents (-5 becomes 0FBh, 255 becomes 0FFh [-1], etc.).

Perhaps a later version of MASM will perform stronger type checking on the values you shove into these variables, perhaps not. However, you should always keep in mind that it will always be possible to circumvent this checking. It's up to you to write your programs correctly. The assembler won't help you as much as Pascal or Ada will. Of course, even if the assembler disallowed these statements, it would still be easy to get around the type checking. Consider the following sequence:

```
                mov     al, -5
                  .
        ; Any number of statements which do not affect AL
                  .
                mov     j, al
```

There is, unfortunately, no way the assembler is going to be able to tell you that you're storing an illegal value into j[5]. The registers, by their very nature, are neither signed nor unsigned. Therefore the assembler will let you store a register into a variable regardless of the value that may be in that register.

Although the assembler does not check to see if both operands to an instruction are signed or unsigned, it most certainly checks their size. If the sizes do not agree the assembler will complain with an appropriate error message. The following examples are all illegal:

```
mov     i, ax       ;Cannot move 16 bits into eight
mov     i, 300      ;300 won't fit in eight bits.
mov     k, -130     ;-130 won't fit into eight bits.
```

You might ask "if the assembler doesn't really differentiate signed and unsigned values, why bother with them? Why not simply use db all the time?" Well, there are two reasons. First, it makes your programs easier to read and understand if you explicitly state (by using *byte* and *sbyte*) which variables are signed and which are unsigned. Second, who said anything about the assembler ignoring whether the variables are signed or unsigned? The mov instruction ignores the difference, but there are other instructions that do not.

One final point is worth mentioning concerning the declaration of byte variables. In all of the declarations you've seen thus far the operand field of the instruction has always contained a question mark. This question mark tells the assembler that the variable should be left uninitialized when DOS loads the program into memory[6]. You may specify an initial value for the variable, that will be loaded into memory before the program starts executing, by replacing the question mark with your initial value. Consider the following byte variable declarations:

```
i               db      0
j               byte    255
k               sbyte   -1
```

In this example, the assembler will initialize i, j, and k to zero, 255, and -1, respectively, when the program loads into memory. This fact will prove quite useful later on, especially when discussing tables and arrays. Once again, the assembler only checks the sizes of the operands. It does not check to make sure that the operand for the byte directive is positive or that the value in the operand field of sbyte is in the range -128…127. MASM will allow any value in the range -128…255 in the operand field of any of these statements.

In case you get the impression that there isn't a real reason to use byte vs. sbyte in a program, you should note that while MASM sometimes ignores the differences in these definitions, Microsoft's CodeView debugger does not. If you've declared a variable as a signed value, CodeView will display it as such (including a minus sign, if necessary). On the other hand, CodeView will always display db and byte variables as positive values.

5.3.2 Declaring and using WORD Variables

Most 80x86 programs use word values for three things: 16 bit signed integers, 16 bit unsigned integers, and offsets (pointers). Oh sure, you can use word values for lots of other things as well, but these three represent most applications of the word data type. Since the word is the largest data type the 8086, 8088, 80186, 80188, and 80286 can handle, you'll find that for most programs, the word is the basis for most computations. Of course, the 80386 and later allow 32 bit computations, but many programs do not use these 32 bit instructions since that would limit them to running on 80386 or later CPUs.

You use the dw, word, and sword statements to declare word variables. The following examples demonstrate their use:

5. Actually, for this simple example you *could* modify the assembler to detect this problem. But it's easy enough to come up with a slightly more complex example where the assembler could *not* detect the problem on.
6. DOS actually initializes such variables to zero, but you shouldn't count on this.

```
NoSignedWord     dw       ?
UnsignedWord     word     ?
SignedWord       sword    ?
Initialized0     word     0
InitializedM1    sword    -1
InitializedBig   word     65535
InitializedOfs   dw       NoSignedWord
```

Most of these declarations are slight modifications of the byte declarations you saw in the last section. Of course you may initialize any word variable to a value in the range -32768…65535 (the union of the range for signed and unsigned 16 bit constants). The last declaration above, however, is new. In this case a label appears in the operand field (specifically, the name of the NoSignedWord variable). When a label appears in the operand field the assembler will substitute the offset of that label (within the variable's segment). If these were the only declarations in *dseg* and they appeared in this order, the last declaration above would initialize *InitializedOfs* with the value zero since *NoSignedWord*'s offset is zero within the data segment. This form of initialization is quite useful for initializing *pointers*. But more on that subject later.

The CodeView debugger differentiates dw/word variables and sword variables. It always displays the unsigned values as positive integers. On the other hand, it will display sword variables as signed values (complete with minus sign, if the value is negative). Debugging support is one of the main reasons you'll want to use word or sword as appropriate.

5.3.3 Declaring and using DWORD Variables

You may use the dd, dword, and sdword instructions to declare four-byte integers, pointers, and other variables types. Such variables will allow values in the range -2,147,483,648…4,294,967,295 (the union of the range of signed and unsigned four-byte integers). You use these declarations like the word declarations:

```
NoSignedDWord    dd       ?
UnsignedDWord    dword    ?
SignedDWord      sdword   ?
InitBig          dword    4000000000
InitNegative     sdword   -1
InitPtr          dd       InitBig
```

The last example initializes a double word pointer with the segment:offset address of the InitBig variable.

Once again, it's worth pointing out that the assembler doesn't check the types of these variables when looking at the initialization values. If the value fits into 32 bits, the assembler will accept it. Size checking, however, is strictly enforced. Since the only 32 bit mov instructions on processors earlier than the 80386 are les and lds, you will get an error if you attempt to access dword variables on these earlier processors using a mov instruction. Of course, even on the 80386 you cannot move a 32 bit variable into a 16 bit register, you must use the 32 bit registers. Later, you'll learn how to manipulate 32 bit variables, even on a 16 bit processor. Until then, just pretend that you can't.

Keep in mind, of course, that CodeView differentiates between dd/dword and sdword. This will help you see the actual values your variables have when you're debugging your programs. CodeView only does this, though, if you use the proper declarations for your variables. Always use sdword for signed values and dd or dword (dword is best) for unsigned values.

5.3.4 Declaring and using FWORD, QWORD, and TBYTE Variables

MASM 6.x also lets you declare six-byte, eight-byte, and ten-byte variables using the df/fword, dq/qword, and dt/tbyte statements. Declarations using these statements were originally intended for floating point and BCD values. There are better directives for the floating point variables and you don't need to concern yourself with the other data types you'd use these directives for. The following discussion is for completeness' sake.

The df/fword statement's main utility is declaring 48 bit pointers for use in 32 bit protected mode on the 80386 and later. Although you could use this directive to create an arbitrary six byte variable, there are better directives for doing that. You should only use this directive for 48 bit far pointers on the 80386.

dq/qword lets you declare *quadword* (eight byte) variables. The original purpose of this directive was to let you create 64 bit double precision floating point variables and 64 bit integer variables. There are better directives for creating floating point variables. As for 64 bit integers, you won't need them very often on the 80x86 CPU (at least, not until Intel releases a member of the 80x86 family with 64 bit general purpose registers).

The dt/tbyte directives allocate ten bytes of storage. There are two data types indigenous to the 80x87 (math coprocessor) family that use a ten byte data type: ten byte BCD values and extended precision (80 bit) floating point values. This text will pretty much ignore the BCD data type. As for the floating point type, once again there is a better way to do it.

5.3.5 Declaring Floating Point Variables with REAL4, REAL8, and REAL10

These are the directives you should use when declaring floating point variables. Like dd, dq, and dt these statements reserve four, eight, and ten bytes. The operand fields for these statements may contain a question mark (if you don't want to initialize the variable) or it may contain an initial value in floating point form. The following examples demonstrate their use:

```
x               real4    1.5
y               real8    1.0e-25
z               real10   -1.2594e+10
```

Note that the operand field must contain a valid floating point constant using either decimal or scientific notation. In particular, *pure integer constants are not allowed.* The assembler will complain if you use an operand like the following:

```
x               real4      1
```

To correct this, change the operand field to "1.0".

Please note that it takes special hardware to perform floating point operations (e.g., an 80x87 chip or an 80x86 with built-in math coprocessor). If such hardware is not available, you must write software to perform operations like floating point addition, subtraction, multiplication, etc. In particular, you cannot use the 80x86 add instruction to add two floating point values. This text will cover floating point arithmetic in a later chapter (see "Floating Point Arithmetic" on page 771). Nonetheless, it's appropriate to discuss how to declare floating point variables in the chapter on data structures.

MASM also lets you use dd, dq, and dt to declare floating point variables (since these directives reserve the necessary four, eight, or ten bytes of space). You can even initialize such variables with floating point constants in the operand field. But there are two major drawbacks to declaring variables this way. First, as with bytes, words, and double words, the CodeView debugger will only display your floating point variables properly if you use the real4, real8, or real10 directives. If you use dd, dq, or dt, CodeView will display your values as four, eight, or ten byte unsigned integers. Another, potentially bigger, problem with using dd, dq, and dt is that they allow both integer and floating point constant initializers (remember, real4, real8, and real10 do not). Now this might seem like a good feature

at first glance. However, the integer representation for the value one is *not* the same as the floating point representation for the value 1.0. So if you accidentally enter the value "1" in the operand field when you really meant "1.0", the assembler would happily digest this and then give you incorrect results. Hence, you should always use the real4, real8, and real10 statements to declare floating point variables.

5.4 Creating Your Own Type Names with TYPEDEF

Let's say that you simply do not like the names that Microsoft decided to use for declaring byte, word, dword, real, and other variables. Let's say that you prefer Pascal's naming convention or, perhaps, C's naming convention. You want to use terms like *integer, float, double, char, boolean,* or whatever. If this were Pascal you could redefine the names in the **type** section of the program. With C you could use a "**#define**" or a **typedef** statement to accomplish the task. Well, MASM 6.x has it's own *typedef* statement that also lets you create aliases of these names. The following example demonstrates how to set up some Pascal compatible names in your assembly language programs:

```
integer         typedef     sword
char            typedef     byte
boolean         typedef     byte
float           typedef     real4
colors          typedef     byte
```

Now you can declare your variables with more meaningful statements like:

```
i               integer     ?
ch              char        ?
FoundIt         boolean     ?
x               float       ?
HouseColor      colors      ?
```

If you are an Ada, C, or FORTRAN programmer (or any other language, for that matter), you can pick type names you're more comfortable with. Of course, this doesn't change how the 80x86 or MASM reacts to these variables one iota, but it does let you create programs that are easier to read and understand since the type names are more indicative of the actual underlying types.

Note that CodeView still respects the underlying data type. If you define *integer* to be an sword type, CodeView will display variables of type integer as signed values. Likewise, if you define *float* to mean real4, CodeView will still properly display *float* variables as four-byte floating point values.

5.5 Pointer Data Types

Some people refer to pointers as scalar data types, others refer to them as composite data types. This text will treat them as scalar data types even though they exhibit some tendencies of both scalar and composite data types (for a complete description of composite data types, see "Composite Data Types" on page 206).

Of course, the place to start is with the question "What is a pointer?" Now you've probably experienced pointers first hand in the Pascal, C, or Ada programming languages and you're probably getting worried right now. Almost everyone has a real bad experience when they first encounter pointers in a high level language. Well, fear not! Pointers are actually *easier* to deal with in assembly language. Besides, most of the problems you had with pointers probably had nothing to do with pointers, but rather with the linked list and tree data structures you were trying to implement with them. Pointers, on the other hand, have lots of uses in assembly language that have nothing to do with linked lists, trees, and other scary data structures. Indeed, simple data structures like arrays and records often involve the use of pointers. So if you've got some deep-rooted fear about

pointers, well forget everything you know about them. You're going to learn how *great* pointers really are.

Probably the best place to start is with the definition of a pointer. Just exactly what is a pointer, anyway? Unfortunately, high level languages like Pascal tend to hide the simplicity of pointers behind a wall of abstraction. This added complexity (which exists for good reason, by the way) tends to frighten programmers because *they don't understand what's going on*.

Now if you're afraid of pointers, well, let's just ignore them for the time being and work with an array. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don't know Pascal, the concept here is pretty easy to understand. M is an array with 1024 integers in it, indexed from M[0] to M[1023]. Each one of these array *elements* can hold an integer value that is independent of all the others. In other words, this array gives you 1024 different integer variables each of which you refer to by number (the array index) rather than by name.

If you encountered a program that had the statement M[0]:=100 you probably wouldn't have to think at all about what is happening with this statement. It is storing the value 100 into the first element of the array M. Now consider the following two statements:

```
i := 0; (* Assume "i" is an integer variable *)
M [i] := 100;
```

You should agree, without too much hesitation, that these two statements perform the same exact operation as M[0]:=100;. Indeed, you're probably willing to agree that you can use any integer expression in the range 0…1023 as an index into this array. The following statements *still* perform the same operation as our single assignment to index zero:

```
i := 5;                 (* assume all variables are integers*)
j := 10;
k := 50;
m [i*j-k] := 100;
```

"Okay, so what's the point?" you're probably thinking. "Anything that produces an integer in the range 0…1023 is legal. So what?" Okay, how about the following:

```
M [1] := 0;
M [ M [1] ] := 100;
```

Whoa! Now that takes a few moments to digest. However, if you take it slowly, it makes sense and you'll discover that these two instructions perform the exact same operation you've been doing all along. The first statement stores zero into array element M[1]. The second statement fetches the value of M[1], which is an integer so you can use it as an array index into M, and uses that value (zero) to control where it stores the value 100.

If you're willing to accept the above as reasonable, perhaps bizarre, but usable nonetheless, then you'll have no problems with pointers. *Because `m[1]` is a pointer!* Well, not really, but if you were to change "M" to "memory" and treat this array as all of memory, this is the exact definition of a pointer.

A pointer is simply a memory location whose value is the address (or index, if you prefer) of some other memory location. Pointers are very easy to declare and use in an assembly language program. You don't even have to worry about array indices or anything like that. In fact, the only complication you're going to run into is that the 80x86 supports two kinds of pointers: *near* pointers and *far* pointers.

A near pointer is a 16 bit value that provides an offset into a segment. It could be any segment but you will generally use the data segment (*dseg* in SHELL.ASM). If you have a word variable p that contains 1000h, then p "points" at memory location 1000h in *dseg*. To access the word that p points at, you could use code like the following:

```
mov     bx, p       ;Load BX with pointer.
mov     ax, [bx]    ;Fetch data that p points at.
```

By loading the value of p into bx this code loads the value 1000h into bx (assuming p contains 1000h and, therefore, points at memory location 1000h in *dseg*). The second instruction above loads the ax register with the word starting at the location whose offset appears in bx. Since bx now contains 1000h, this will load ax from locations DS:1000 and DS:1001.

Why not just load ax directly from location 1000h using an instruction like mov ax,ds:[1000h]? Well, there are lots of reasons. But the primary reason is that this single instruction *always* loads ax from location 1000h. Unless you are willing to mess around with self-modifying code, you cannot change the location from which it loads ax. The previous two instructions, however, always load ax from the location that p points at. This is very easy to change under program control, without using self-modifying code. In fact, the simple instruction mov p,2000h will cause those two instructions above to load ax from memory location DS:2000 the next time they execute. Consider the following instructions:

```
        lea     bx, i       ;This can actually be done with
        mov     p, bx       ; a single instruction as you'll
         .                  ; see in Chapter Eight.
         .
< Some code that skips over the next two instructions >

        lea     bx, j       ;Assume the above code skips these
        mov     p, bx       ; two instructions, that you get
         .                  ; here by jumping to this point from
         .                  ; somewhere else.
        mov     bx, p       ;Assume both code paths above wind
        mov     ax, [bx]    ; up down here.
```

This short example demonstrates two execution paths through the program. The first path loads the variable p with the address of the variable i (remember, lea loads bx with the offset of the second operand). The second path through the code loads p with the address of the variable j. Both execution paths converge on the last two mov instructions that load ax with i or j depending upon which execution path was taken. In many respects, this is like a *parameter* to a procedure in a high level language like Pascal. Executing the same instructions accesses different variables depending on whose address (i or j) winds up in p.

Sixteen bit near pointers are small, fast, and the 80x86 provides efficient access using them. Unfortunately, they have one very serious drawback – you can only access 64K of data (one segment) when using near pointers[7]. Far pointers overcome this limitation at the expense of being 32 bits long. However, far pointers let you access any piece of data anywhere in the memory space. For this reason, and the fact that the UCR Standard Library uses far pointers exclusively, this text will use far pointers most of the time. But keep in mind that this is a decision based on trying to keep things simple. Code that uses near pointers rather than far pointers will be shorter and faster.

To access data referenced by a 32 bit pointer, you will need to load the offset portion (L.O. word) of the pointer into bx, bp, si, or di and the segment portion into a segment register (typically es). Then you could access the object using the register indirect addressing mode. Since the les instruction is so convenient for this operation, it is the perfect choice for loading es and one of the above four registers with a pointer value. The following sample code stores the value in al into the byte pointed at by the far pointer p:

```
        les     bx, p       ;Load p into ES:BX
        mov     es:[bx], al ;Store away AL
```

Since near pointers are 16 bits long and far pointers are 32 bits long, you could simply use the dw/word and dd/dword directives to allocate storage for your pointers (pointers are inherently unsigned, so you wouldn't normally use sword or sdword to declare a pointer).

7. Technically, this isn't true. A single pointer is limited to accessing data in one particular segment at a time, but you could have several near pointers each pointing at data in different segments. Unfortunately, you need to keep track of all this yourself and it gets out of hand very quickly as the number of pointers in your program increases.

However, there is a much better way to do this by using the typedef statement. Consider the following general forms:

```
typename        typedef   near ptr basetype
typename        typedef   far ptr basetype
```

In these two examples *typename* represents the name of the new type you're creating while *basetype* is the name of the type you want to create a pointer for. Let's look at some specific examples:

```
nbytptr         typedef   near ptr byte
fbytptr         typedef   far ptr byte
colorsptr       typedef   far ptr colors
wptr            typedef   near ptr word
intptr          typedef   near ptr integer
intHandle       typedef   near ptr intptr
```

(these declarations assume that you've previously defined the types *colors* and *integer* with the typedef statement). The typedef statements with the *near ptr* operands produce 16 bit near pointers. Those with the *far ptr* operands produce 32 bit far pointers. MASM 6.x ignores the base type supplied after the *near ptr* or *far ptr*. However, CodeView uses the base type to display the object a pointer refers to in its correct format.

Note that you can use *any* type as the base type for a pointer. As the last example above demonstrates, you can even define a pointer to another pointer (a *handle*). CodeView would properly display the object a variable of type intHandle points at as an address.

With the above types, you can now generate pointer variables as follows:

```
bytestr         nbytptr     ?
bytestr2        fbytptr     ?
CurrentColor    colorsptr   ?
CurrentItem     wptr        ?
LastInt         intptr      ?
```

Of course, you can initialize these pointers at assembly time if you know where they are going to point when the program first starts running. For example, you could initialize the *bytestr* variable above with the offset of *MyString* using the following declaration:

```
bytestr         nbytptr   MyString
```

5.6 Composite Data Types

Composite data types are those that are built up from other (generally scalar) data types. An array is a good example of a composite data type – it is an aggregate of elements all the same type. Note that a composite data type need not be composed of scalar data types, there are arrays of arrays for example, but ultimately you can decompose a composite data type into some primitive, scalar, types.

This section will cover two of the more common composite data types: arrays and records. It's a little premature to discuss some of the more advanced composite data types.

5.6.1 Arrays

Arrays are probably the most commonly used composite data type. Yet most beginning programmers have a very weak understanding of how arrays operate and their associated efficiency trade-offs. It's surprising how many novice (and even advanced!) programmers view arrays from a completely different perspective once they learn how to deal with arrays at the machine level.

Abstractly, an array is an aggregate data type whose members (elements) are all the same type. Selection of a member from the array is by an integer index[8]. Different indices select unique elements of the array. This text assumes that the integer indices are contigu-

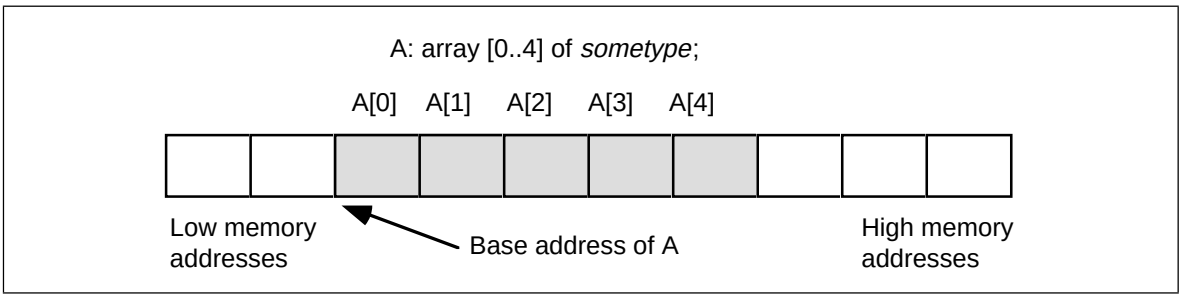


Figure 5.1 Single Dimension Array Implementation

ous (though it is by no means required). That is, if the number x is a valid index into the array and y is also a valid index, with x < y, then all i such that x < i < y are valid indices into the array.

Whenever you apply the indexing operator to an array, the result is the specific array element chosen by that index. For example, A[i] chooses the ith element from array A. Note that there is no formal requirement that element i be anywhere near element i+1 in memory. As long as A[i] always refers to the same memory location and A[i+1] always refers to its corresponding location (and the two are different), the definition of an array is satisfied.

In this text, arrays occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 5.1.

The *base address* of an array is the address of the first element on the array and always appears in the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, etc. Note that there is no requirement that the indices start at zero. They may start with any number as long as they are contiguous. However, for the purposes of discussion, it's easier to discuss accessing array elements if the first index is zero. This text generally begins most arrays at index zero unless there is a good reason to do otherwise. However, this is for consistency only. There is no efficiency benefit one way or another to starting the array index at some value other than zero.

To access an element of an array, you need a function that converts an array index into the address of the indexed element. For a single dimension array, this function is very simple. It is

```
Element_Address = Base_Address + ((Index - Initial_Index) * Element_Size)
```

where Initial_Index is the value of the first index in the array (which you can ignore if zero) and the value Element_Size is the size, in bytes, of an individual element of the array.

5.6.1.1 Declaring Arrays in Your Data Segment

Before you access elements of an array, you need to set aside storage for that array. Fortunately, array declarations build on the declarations you've seen so far. To allocate *n* elements in an array, you would use a declaration like the following:

```
arrayname       basetype    n dup (?)
```

Arrayname is the name of the array variable and *basetype* is the type of an element of that array. This sets aside storage for the array. To obtain the base address of the array, just use *arrayname*.

The n dup (?) operand tells the assembler to duplicate the object inside the parentheses *n* times. Since a question mark appears inside the parentheses, the definition above

8. Or some value whose underlying representation is integer, such as character, enumerated, and boolean types.

would create *n* occurrences of an uninitialized value. Now let's look at some specific examples:

```
CharArray       char     128 dup (?)          ;array[0..127] of char
IntArray        integer  8 dup (?)            ;array[0..7] of integer
BytArray        byte     10 dup (?)           ;array[0..9] of byte
PtrArray        dword    4 dup (?)            ;array[0..3] of dword
```

The first two examples, of course, assume that you've used the typedef statement to define the char and integer data types.

These examples all allocate storage for uninitialized arrays. You may also specify that the elements of the arrays be initialized to a single value using declarations like the following:

```
RealArray       real4    8 dup (1.0)
IntegerAry      integer  8 dup (1)
```

These definitions both create arrays with eight elements. The first definition initializes each four-byte real value to 1.0, the second declaration initializes each integer element to one.

This initialization mechanism is fine if you want each element of the array to have the same value. What if you want to initialize each element of the array with a (possibly) different value? Well, that is easily handled as well. The variable declaration statements you've seen thus far offer yet another initialization form:

name *type* $value_1$, $value_2$, $value_3$, …, $value_n$

This form allocates *n* variables of type *type*. It initializes the first item to $value_1$, the second item to $value_2$, etc. So by simply enumerating each value in the operand field, you can create an array with the desired initial values. In the following integer array, for example, each element contains the square of its index:

```
Squares         integer  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

If your array has more elements than will fit on one line, there are several ways to continue the array onto the next line. The most straight-forward method is to use another integer statement *but without a label*:

```
Squares         integer  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
                integer  121, 144, 169, 196, 225, 256, 289, 324
                integer  361, 400
```

Another option, that is better in some circumstances, is to use a backslash at the end of each line to tell MASM 6.x to continue reading data on the next line:

```
Squares         integer  0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, \
                         121, 144, 169, 196, 225, 256, 289, 324, \
                         361, 400
```

Of course, if your array has several thousand elements in it, typing them all in will not be very much fun. Most arrays initialized this way have no more than a couple hundred entries, and generally far less than 100.

You need to learn about one final technique for initializing single dimension arrays before moving on. Consider the following declaration:

```
BigArray        word     256 dup (0,1,2,3)
```

This array has 1024 elements, not 256. The n dup (xxxx) operand tells MASM to duplicate xxxx *n* times, not create an array with *n* elements. If xxxx consists of a single item, then the dup operator will create an *n* element array. However, if xxxx contains two items separated by a comma, the dup operator will create an array with 2**n* elements. If xxxx contains three items separated by commas, the dup operator creates an array with 3**n* items, and so on. Since there are four items in the parentheses above, the dup operator creates 256*4 or 1024 items in the array. The values in the array will initially be 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 ...

You will see some more possibilities with the dup operator when looking at multidimensional arrays a little later.

5.6.1.2 Accessing Elements of a Single Dimension Array

To access an element of a zero-based array, you can use the simplified formula:

Element_Address = Base_Address + index * Element_Size

For the Base_Address entry you can use the name of the array (since MASM associates the address of the first operand with the label). The Element_Size entry is the number of bytes for each array element. If the object is an array of bytes, the Element_Size field is one (resulting in a very simple computation). If each element of the array is a word (or integer, or other two-byte type) then Element_Size is two. And so on. To access an element of the Squares array in the previous section, you'd use the formula:

Element_Address = Squares + index*2

The 80x86 code equivalent to the statement AX:=Squares[index] is

```
mov     bx, index
add     bx, bx                  ;Sneaky way to compute 2*bx
mov     ax, Squares [bx]
```

There are two important things to notice here. First of all, this code uses the add instruction rather than the mul instruction to compute 2*index. The main reason for choosing add is that it was more convenient (remember, mul doesn't work with constants and it only operates on the ax register). It turns out that add is a *lot* faster than mul on many processors, but since you probably didn't know that, it wasn't an overriding consideration in the choice of this instruction.

The second thing to note about this instruction sequence is that it does not explicitly compute the sum of the base address plus the index times two. Instead, it relies on the indexed addressing mode to implicitly compute this sum. The instruction mov ax, Squares[bx] loads ax from location Squares+bx which is the base address plus index*2 (since bx contains index*2). Sure, you could have used

```
lea     ax, Squares
add     bx, ax
mov     ax, [bx]
```

in place of the last instruction, but why use three instructions where one will do the same job? This is a good example of why you should know your addressing modes inside and out. Choosing the proper addressing mode can reduce the size of your program, thereby speeding it up.

The indexed addressing mode on the 80x86 is a natural for accessing elements of a single dimension array. Indeed, it's syntax even suggests an array access. The only thing to keep in mind is that you must remember to multiply the index by the size of an element. Failure to do so will produce incorrect results.

If you are using an 80386 or later, you can take advantage of the scaled indexed addressing mode to speed up accessing an array element even more. Consider the following statements:

```
mov     ebx, index              ;Assume a 32 bit value.
mov     ax, Squares [ebx*2]
```

This brings the instruction count down to two instructions. You'll soon see that two instructions aren't necessarily faster than three instructions, but hopefully you get the idea. Knowing your addressing modes can surely help.

Before moving on to multidimensional arrays, a couple of additional points about addressing modes and arrays are in order. The above sequences work great if you only access a single element from the Squares array. However, if you access several different elements from the array within a short section of code, and you can afford to dedicate

another register to the operation, you can certainly shorten your code and, perhaps, speed it up as well. The mov ax,Squares[BX] instruction is four bytes long (assuming you need a two-byte displacement to hold the offset to Squares in the data segment). You can reduce this to a two byte instruction by using the base/indexed addressing mode as follows:

```
lea     bx, Squares
mov     si, index
add     si, si
mov     ax, [bx][si]
```

Now bx contains the base address and si contains the index*2 value. Of course, this just replaced a single four-byte instruction with a three-byte and a two-byte instruction, hardly a good trade-off. However, you do not have to reload bx with the base address of Squares for the next access. The following sequence is one byte shorter than the comparable sequence that doesn't load the base address into bx:

```
lea     bx, Squares
mov     si, index
add     si, si
mov     ax, [bx][si]
 .
 .                  ;Assumption: BX is left alone
 .                  ; through this code.
mov     si, index2
add     si, si
mov     cx, [bx][si]
```

Of course the more accesses to Squares you make without reloading bx, the greater your savings will be. Tricky little code sequences such as this one sometimes pay off handsomely. However, the savings depend entirely on which processor you're using. Code sequences that run faster on an 8086 might actually run *slower* on an 80486 (and vice versa). Unfortunately, if speed is what you're after there are no hard and fast rules. In fact, it is very difficult to predict the speed of most instructions on the simple 8086, even more so on processors like the 80486 and Pentium/80586 that offer pipelining, on-chip caches, and even superscalar operation.

5.6.2 Multidimensional Arrays

The 80x86 hardware can easily handle single dimension arrays. Unfortunately, there is no magic addressing mode that lets you easily access elements of multidimensional arrays. That's going to take some work and lots of instructions.

Before discussing how to declare or access multidimensional arrays, it would be a good idea to figure out how to implement them in memory. The first problem is to figure out how to store a multi-dimensional object into a one-dimensional memory space.

Consider for a moment a Pascal array of the form A:array[0..3,0..3] of char. This array contains 16 bytes organized as four rows of four characters. Somehow you've got to draw a correspondence with each of the 16 bytes in this array and 16 contiguous bytes in main memory. Figure 5.2 shows one way to do this.

The actual mapping is not important as long as two things occur: (1) each element maps to a unique memory location (that is, no two entries in the array occupy the same memory locations) and (2) the mapping is consistent. That is, a given element in the array always maps to the same memory location. So what you really need is a function with two input parameters (row and column) that produces an offset into a linear array of sixteen bytes.

Now any function that satisfies the above constraints will work fine. Indeed, you could randomly choose a mapping as long as it was unique. However, what you really want is a mapping that is efficient to compute at run time and works for any size array (not just 4x4 or even limited to two dimensions). While there are a large number of possi-

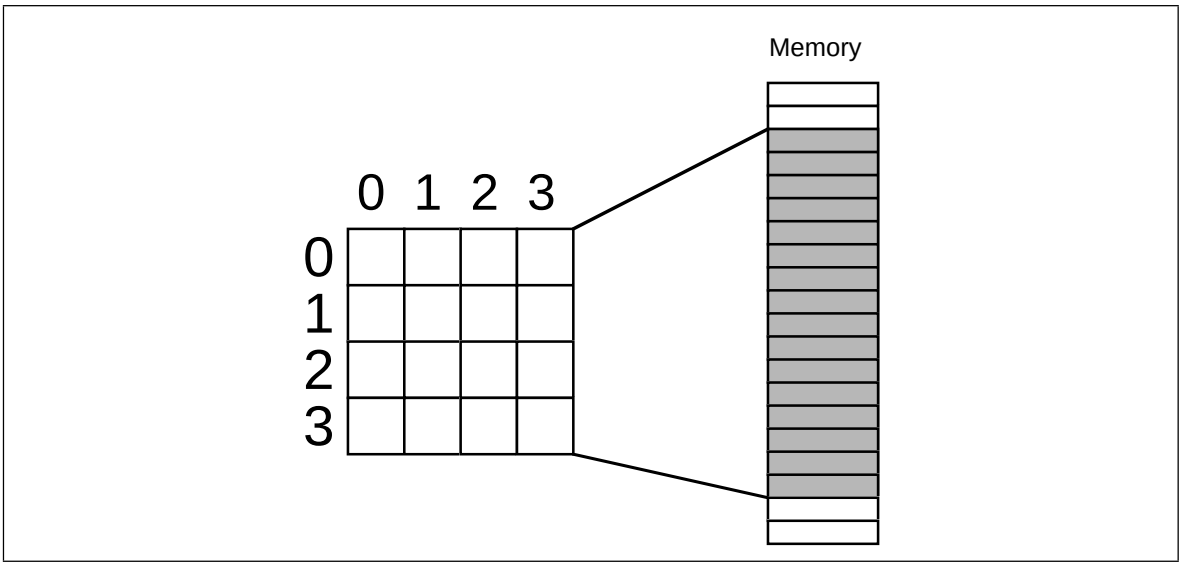


Figure 5.2 Mapping a 4 x 4 Array to Memory

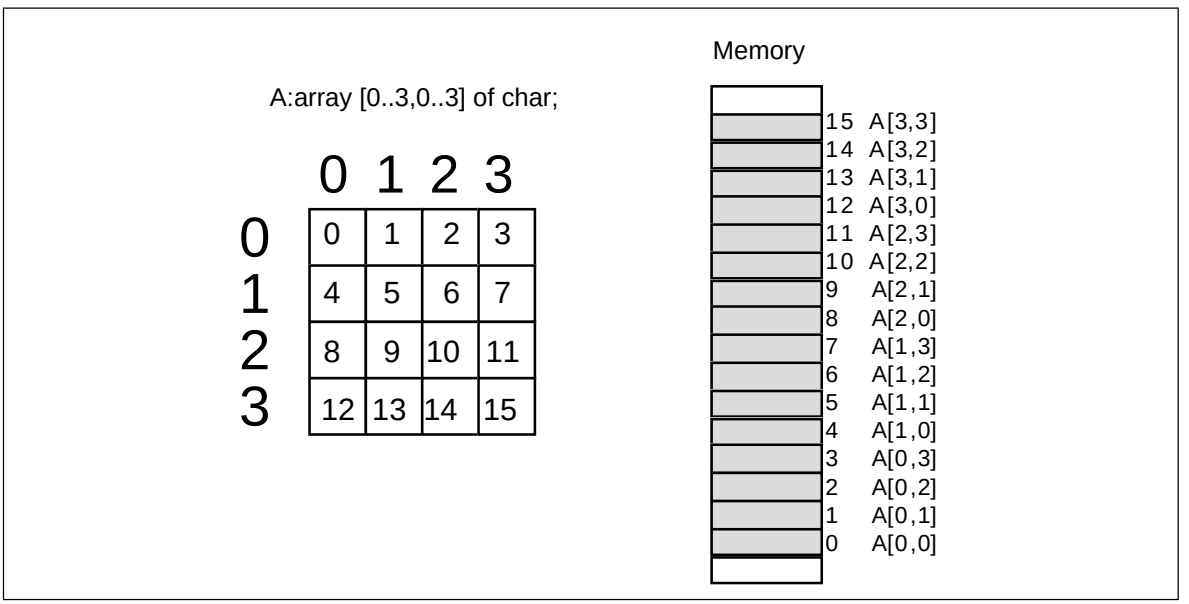


Figure 5.3 Row Major Element Ordering

ble functions that fit this bill, there are two functions in particular that most programmers and most high level languages use: *row major ordering* and *column major ordering.*

5.6.2.1 Row Major Ordering

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. The mapping is best described in Figure 5.3.

Row major ordering is the method employed by most high level programming languages including Pascal, C, Ada, Modula-2, etc. It is very easy to implement and easy to use in machine language (especially within a debugger such as CodeView). The conversion from a two-dimensional structure to a linear array is very intuitive. You start with the

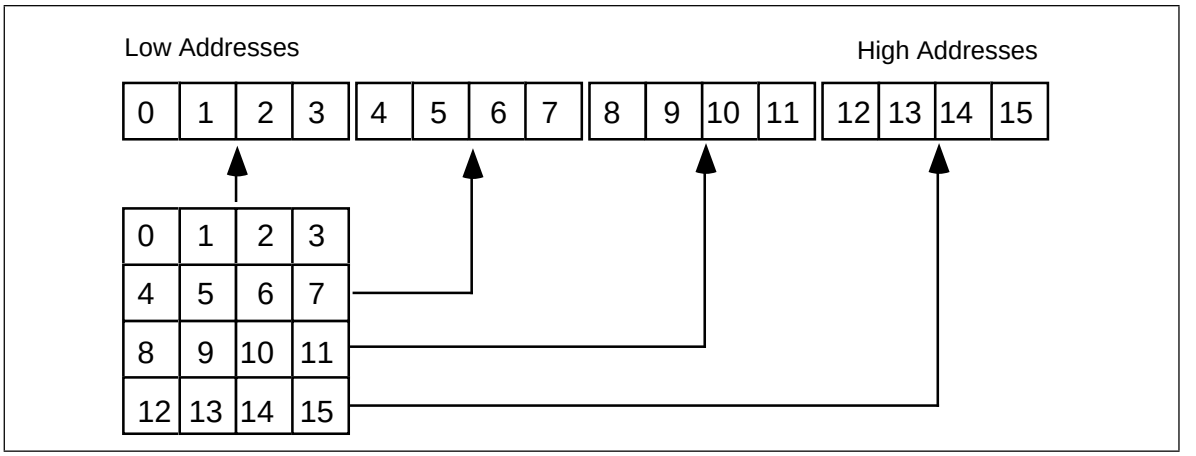


Figure 5.4 Another View of Row Major Ordering for a 4x4 Array

first row (row number zero) and then concatenate the second row to its end. You then concatenate the third row to the end of the list, then the fourth row, etc. (see Figure 5.4).

For those who like to think in terms of program code, the following nested Pascal loop also demonstrates how row major ordering works:

```
index := 0;
for colindex := 0 to 3 do
       for rowindex := 0 to 3 do
       begin
                memory [index] := rowmajor [colindex][rowindex];
                index := index + 1;
       end;
```

The important thing to note from this code, that applies across the board to row major order no matter how many dimensions it has, is that the rightmost index increases the fastest. That is, as you allocate successive memory locations you increment the rightmost index until you reach the end of the current row. Upon reaching the end, you reset the index back to the beginning of the row and increment the next successive index by one (that is, move down to the next row.). This works equally well for any number of dimensions[9]. The following Pascal segment demonstrates row major organization for a 4x4x4 array:

```
index := 0;
for depthindex := 0 to 3 do
       for colindex := 0 to 3 do
          for rowindex := 0 to 3 do begin
                memory [index] := rowmajor [depthindex][colindex][rowindex];
                index := index + 1;
          end;
```

The actual function that converts a list of index values into an offset doesn't involve loops or much in the way of fancy computations. Indeed, it's a slight modification of the formula for computing the address of an element of a single dimension array. The formula to compute the offset for a two-dimension row major ordered array declared as A:array [0..3,0..3] of integer is

```
Element_Address = Base_Address + (colindex * row_size + rowindex) * Element_Size
```

As usual, Base_Address is the address of the first element of the array (A[0][0] in this case) and Element_Size is the size of an individual element of the array, in bytes. Colindex is the leftmost index, rowindex is the rightmost index into the array. Row_size is the number of

9. By the way, the number of dimensions of an array is its *arity*.

elements in one row of the array (four, in this case, since each row has four elements). Assuming Element_Size is one, This formula computes the following offsets from the base address:

Column Index	Row Index	Offset into Array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

For a three-dimensional array, the formula to compute the offset into memory is the following:

```
Address = Base + ((depthindex*col_size+colindex) * row_size + rowindex) *
Element_Size
```

Col_size is the number of items in a column, row_size is the number of items in a row. In Pascal, if you've declared the array as "A:array [i..j] [k..l] [m..n] of *type*;" then row_size is equal to n-m+1 and col_size is equal to l-k+1.

For a four dimensional array, declared as "A:array [g..h] [i..j] [k..l] [m..n] of *type*;" the formula for computing the address of an array element is

```
Address =
Base + (((LeftIndex * depth_size + depthindex)*col_size+colindex) * row_size +
rowindex) * Element_Size
```

Depth_size is equal to i-j+1, col_size and row_size are the same as before. LeftIndex represents the value of the leftmost index.

By now you're probably beginning to see a pattern. There is a generic formula that will compute the offset into memory for an array with *any* number of dimensions, however, you'll rarely use more than four.

Another convenient way to think of row major arrays is as arrays of arrays. Consider the following *single dimension* array definition:

```
A: array [0..3] of sometype;
```

Assume that *sometype* is the type "sometype = array [0..3] of char;".

A is a single dimension array. Its individual elements happen to be arrays, but you can safely ignore that for the time being. The formula to compute the address of an element of a single dimension array is

```
Element_Address = Base + Index * Element_Size
```

In this case Element_Size happens to be four since each element of A is an array of four characters. So what does this formula compute? It computes the base address of each row in this 4x4 array of characters (see Figure 5.5).

Of course, once you compute the base address of a row, you can reapply the single dimension formula to get the address of a particular element. While this doesn't affect the computation at all, conceptually it's probably a little easier to deal with several single dimension computations rather than a complex multidimensional array element address computation.

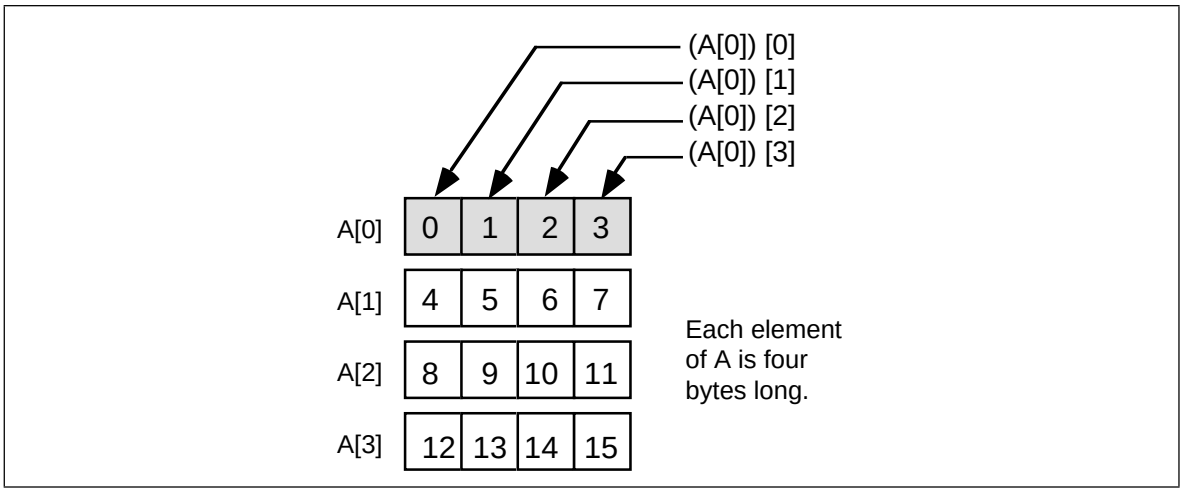


Figure 5.5 Viewing a 4x4 Array as an Array of Arrays

Consider a Pascal array defined as "A:array [0..3] [0..3] [0..3] [0..3] [0..3] of char;" You can view this five-dimension array as a single dimension array of arrays:

```
type
                OneD = array [0..3] of char;
                TwoD = array [0..3] of OneD;
                ThreeD = array [0..3] of TwoD;
                FourD = array [0..3] of ThreeD;
var
                A : array [0..3] of FourD;
```

The size of OneD is four bytes. Since TwoD contains four OneD arrays, its size is 16 bytes. Likewise, ThreeD is four TwoDs, so it is 64 bytes long. Finally, FourD is four ThreeDs, so it is 256 bytes long. To compute the address of "A [b] [c] [d] [e] [f]" you could use the following steps:

- Compute the address of A [b] as "Base + b * size". Here size is 256 bytes. Use this result as the new base address in the next computation.
- Compute the address of A [b] [c] by the formula "Base + c*size", where Base is the value obtained immediately above and size is 64. Use the result as the new base in the next computation.
- Compute the address of A [b] [c] [d] by "Base + d*size" with Base coming from the above computation and size being 16.
- Compute the address of A [b] [c] [d] [e] with the formula "Base + e*size" with Base from above and size being four. Use this value as the base for the next computation.
- Finally, compute the address of A [b] [c] [d] [e] [f] using the formula "Base + f*size" where base comes from the above computation and size is one (obviously you can simply ignore this final multiplication). The result you obtain at this point is the address of the desired element.

Not only is this scheme easier to deal with than the fancy formulae from above, but it is easier to compute (using a single loop) as well. Suppose you have two arrays initialized as follows

A1 = {256, 64, 16, 4, 1} and A2 = {b, c, d, e, f}

then the Pascal code to perform the element address computation becomes:

```
for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

Presumably base contains the base address of the array before executing this loop. Note that you can easily extend this code to any number of dimensions by simply initializing A1 and A2 appropriately and changing the ending value of the for loop.

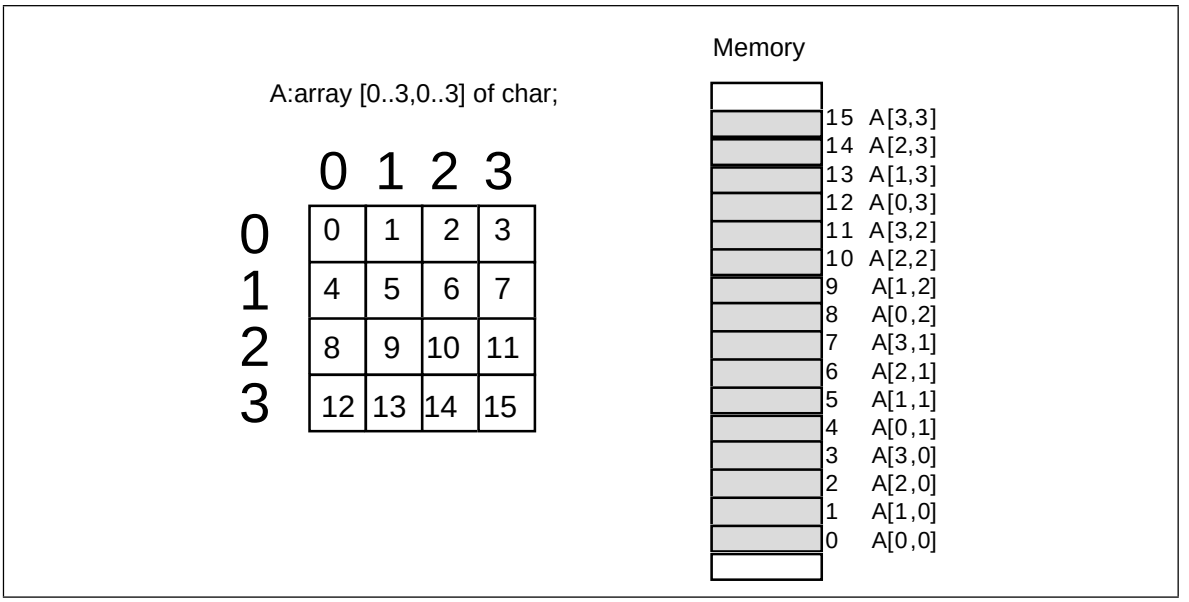


Figure 5.6 Column Major Element Ordering

As it turns out, the computational overhead for a loop like this is too great to consider in practice. You would only use an algorithm like this if you needed to be able to specify the number of dimensions at run time. Indeed, one of the main reasons you won't find higher dimension arrays in assembly language is that assembly language displays the inefficiencies associated with such access. It's easy to enter something like "A [b,c,d,e,f]" into a Pascal program, not realizing what the compiler is doing with the code. Assembly language programmers are not so cavalier – they see the mess you wind up with when you use higher dimension arrays. Indeed, good assembly language programmers try to avoid two dimension arrays and often resort to tricks in order to access data in such an array when its use becomes absolutely mandatory. But more on that a little later.

5.6.2.2 Column Major Ordering

Column major ordering is the other function frequently used to compute the address of an array element. FORTRAN and various dialects of BASIC (e.g., Microsoft) use this method to index arrays.

In row major ordering the rightmost index increased the fastest as you moved through consecutive memory locations. In column major ordering the leftmost index increases the fastest. Pictorially, a column major ordered array is organized as shown in Figure 5.6.

The formulae for computing the address of an array element when using column major ordering is very similar to that for row major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimension column major array:

```
Element_Address = Base_Address + (rowindex * col_size + colindex) * Element_Size
```

For a three-dimension column major array:

```
Address = Base + ((rowindex*col_size+colindex) * depth_size + depthindex) *
Element_Size
```

For a four-dimension column major array:

```
Address = Base + (((rowindex * col_size + colindex)*depth_size+depthindex) *
Left_size + Leftindex) * Element_Size
```

The single Pascal loop provided for row major access remains unchanged (to access A [b] [c] [d] [e] [f]):

```
for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

Likewise, the initial values of the *A1* array remain unchanged:

```
A1 = {256, 64, 16, 4, 1}
```

The only thing that needs to change is the initial values for the *A2* array, and all you have to do here is reverse the order of the indices:

```
A2 = {f, e, d, c, b}
```

5.6.2.3 Allocating Storage for Multidimensional Arrays

If you have an *m* x *n* array, it will have *m* * *n* elements and require *m*n*Element_Size* bytes of storage. To allocate storage for an array you must reserve this amount of memory. As usual, there are several different ways of accomplishing this task. This text will try to take the approach that is easiest to read and understand in your programs.

Reconsider the dup operator for reserving storage. n dup (xxxx) replicates xxxx *n* times. As you saw earlier, this dup operator allows not just one, but several items within the parentheses and it duplicates everything inside the specified number of times. In fact, the dup operator allows *anything* that you might normally expect to find in the operand field of a byte statement *including additional occurrences of the DUP operator.* Consider the following statement:

```
A               byte    4 dup (4 dup (?))
```

The first dup operator repeats everything inside the parentheses four times. Inside the parentheses the 4 DUP (?) operation tells MASM to set aside storage for four bytes. Four copies of four bytes yields 16 bytes, the number necessary for a 4 x 4 array. Of course, to reserve storage for this array you could have just as easily used the statement:

```
A               byte    16 dup (?)
```

Either way the assembler is going to set aside 16 contiguous bytes in memory. As far as the 80x86 is concerned, there is no difference between these two forms. On the other hand, the former version provides a better indication that A is a 4 x 4 array than the latter version. The latter version looks like a single dimension array with 16 elements.

You can very easily extend this concept to arrays of higher arity as well. The declaration for a three dimension array, A:array [0..2, 0..3, 0..4] of integer might be

```
A               integer  3 dup (4 dup (5 dup (?)))
```

(of course, you will need the integer typedef word statement in your program for this to work.)

As was the case with single dimension arrays, you may initialize every element of the array to a specific value by replacing the question mark (?) with some particular value. For example, to initialize the above array so that each element contains one you'd use the code:

```
A               integer  3 dup (4 dup (5 dup (1)))
```

If you want to initialize each element of the array to a different value, you'll have to enter each value individually. If the size of a row is small enough, the best way to approach this task is to place the data for each row of an array on its own line. Consider the following 4x4 array declaration:

```
A               integer  0,1,2,3
                integer    1,0,1,1
                integer    5,7,2,2
                integer    0,0,7,6
```

Once again, the assembler doesn't care where you split the lines, but the above is much easier to identify as a 4x4 array than the following that emits the exact same data:

```
A               integer  0,1,2,3,1,0,1,1,5,7,2,2,0,0,7,6
```

Of course, if you have a large array, an array with really large rows, or an array with many dimensions, there is little hope for winding up with something reasonable. That's when comments that carefully explain everything come in handy.

5.6.2.4 Accessing Multidimensional Array Elements in Assembly Language

Well, you've seen the formulae for computing the address of an array element. You've even looked at some Pascal code you could use to access elements of a multidimensional array. Now it's time to see how to access elements of those arrays using assembly language.

The mov, add, and mul instructions make short work of the various equations that compute offsets into multidimensional arrays. Let's consider a two dimension array first:

```
; Note: TwoD's row size is 16 bytes.

TwoD            integer  4 dup (8 dup (?))
i               integer  ?
j               integer  ?

                 .       .
                 .       .
                 .       .

; To peform the operation TwoD[i,j] := 5; you'd use the code:

                mov      ax, 8        ;8 elements per row
                mul      i
                add      ax, j
                add      ax, ax       ;Multiply by element size (2)
                mov      bx, ax       ;Put in a register we can use
                mov      TwoD [bx], 5
```

Of course, if you have an 80386 chip (or better), you could use the following code[10]:

```
                mov      eax, 8       ;Zeros H.O. 16 bits of EAX.
                mul      i
                add      ax, j
                mov      TwoD[eax*2], 5
```

Note that this code does *not* require the use of a two register addressing mode on the 80x86. Although an addressing mode like TwoD [bx][si] looks like it should be a natural for accessing two dimensional arrays, that isn't the purpose of this addressing mode.

Now consider a second example that uses a three dimension array:

```
ThreeD          integer  4 dup (4 dup (4 dup (?)))
i               integer  ?
j               integer  ?
k               integer  ?

                 .       .
                 .       .
                 .       .

; To peform the operation ThreeD[i,j,k] := 1; you'd use the code:

                mov      bx, 4        ;4 elements per column
                mov      ax, i
                mul      bx
                add      ax, j
```

10. Actually, there is an even *better* 80386 instruction sequence than this, but it uses instructions yet to be discussed.

```
mul     bx          ;4 elements per row
add     ax, k
add     ax, ax      ;Multiply by element size (2)
mov     bx, ax      ;Put in a register we can use
mov     ThreeD [bx], 1
```

Of course, if you have an 80386 or better processor, this can be improved somewhat by using the following code:

```
mov     ebx, 4
mov     eax, ebx
mul     i
add     ax, j
mul     bx
add     k
mov     ThreeD[eax*2], 1
```

5.6.3 Structures

The second major composite data structure is the Pascal *record* or C *structure*[11]. The Pascal terminology is probably better, since it tends to avoid confusion with the more general term *data structure*. However, MASM uses "structure" so it doesn't make sense to deviate from this. Furthermore, MASM uses the term *record* to denote something slightly different, furthering the reason to stick with the term structure.

Whereas an array is homogeneous, whose elements are all the same, the elements in a structure can be of any type. Arrays let you select a particular element via an integer index. With structures, you must select an element (known as a *field*) by name.

The whole purpose of a structure is to let you encapsulate different, but logically related, data into a single package. The Pascal record declaration for a student is probably the most typical example:

```
student = record
                Name: string [64];
                Major: integer;
                SSN:   string[11];
                Midterm1: integer;
                Midterm2: integer;
                Final: integer;
                Homework: integer;
                Projects: integer;
        end;
```

Most Pascal compilers allocate each field in a record to contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name[12], the next two bytes hold the major code, the next 12 the Social Security Number, etc.

In assembly language, you can also create structure types using the MASM struct statement. You would encode the above record in assembly language as follows:

```
student         struct
Name            char     65 dup (?)
Major           integer  ?
SSN             char     12 dup (?)
Midterm1        integer  ?
Midterm2        integer  ?
Final           integer  ?
Homework        integer  ?
Projects        integer  ?
student         ends
```

11. It also goes by some other names in other languages, but most people recognize at least one of these names.
12. Strings require an extra byte, in addition to all the characters in the string, to encode the length.

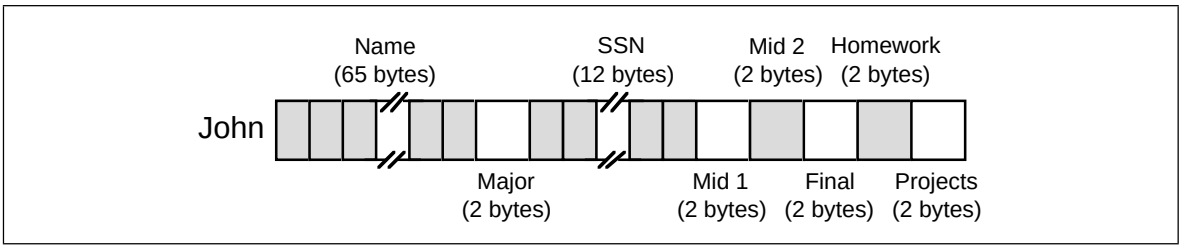


Figure 5.7 Student Data Structure Storage in Memory

Note that the structure ends with the ends (for *end structure*) statement. The label on the ends statement must be the same as on the struct statement.

The field names within the structure must be unique. That is, the same name may not appear two or more times in the same structure. However, all field names are local to that structure. Therefore, you may reuse those field names elsewhere in the program[13].

The struct directive only defines a structure type. It does *not* reserve storage for a structure variable. To actually reserve storage you need to declare a variable using the structure name as a MASM statement, e.g.,

```
John            student     {}
```

The braces must appear in the operand field. Any initial values must appear between the braces. The above declaration allocates memory as shown in Figure 5.7. :

If the label John corresponds to the *base address* of this structure, then the Name field is at offset John+0, the Major field is at offset John+65, the SSN field is at offset John+67, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the Major field in the variable John is at offset 65 from the base address of John. Therefore, you could store the value in ax into this field using the instruction mov John[65], ax. Unfortunately, memorizing all the offsets to fields in a structure defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a structure?

Well, as it turns out, MASM lets you refer to field names in a structure using the same mechanism C and Pascal use: the dot operator. To store ax into the Major field, you could use mov John.Major,ax instead of the previous instruction. This is much more readable and certainly easier to use.

Note that the use of the dot operator does *not* introduce a new addressing mode. The instruction mov John.Major,ax still uses the displacement only addressing mode. MASM simply adds the base address of John with the offset to the Major field (65) to get the actual displacement to encode into the instruction.

You may also specify default initial values when creating a structure. In the previous example, the fields of the student structure were given indeterminate values by specifying "?" in the operand field of each field's declaration. As it turns out, there are two different ways to specify an initial value for structure fields. Consider the following definition of a "point" data structure:

```
Point           struct
x               word    0
y               word    0
z               word    0
Point           ends
```

Whenever you declare a variable of type point using a statement similar to

```
CurPoint        Point   {}
```

13. You *cannot* redefine a fieldname as an equate or macro label. You may, however, reuse a field name as a statement label. Also, note that versions of MASM prior to 6.0 do not support the ability to reuse structure field names.

MASM automatically initializes the CurPoint.x, CurPoint.y, and CurPoint.z variables to zero. This works out great in those cases where your objects usually start off with the same initial values[14]. Of course, it might turn out that you would like to initialize the X, Y, and Z fields of the points you declare, but you want to give each point a different value. That is easily accomplished by specifying initial values inside the braces:

```
Point1          point   {0,1,2}
Point2          point   {1,1,1}
Point3          point   {0,1,1}
```

MASM fills in the values for the fields in the order that they appear in the operand field. For Point1 above, MASM initializes the X field with zero, the Y field with one, and the Z field with two.

The type of the initial value in the operand field must match the type of the corresponding field in the structure definition. You cannot, for example, specify an integer constant for a real4 field, nor could you specify a value greater than 255 for a byte field.

MASM does not require that you initialize all fields in a structure. If you leave a field blank, MASM will use the specified default value (undefined if you specify "?" rather than a default value).

5.6.4 Arrays of Structures and Arrays/Structures as Structure Fields

Structs may contain other structures or arrays as fields. Consider the following definition:

```
Pixel           struct
Pt              point   {}
Color           dword   ?
Pixel           ends
```

The definition above defines a single point with a 32 bit color component. When initializing an object of type Pixel, the first initializer corresponds to the Pt field, *not the x-coordinate field*. **The following definition is incorrect:**

```
ThisPt          Pixel   {5,10}
```

The value of the first field ("5") is not an object of type point. Therefore, the assembler generates an error when encountering this statement. MASM will allow you to initialize the fields of ThisPt using declarations like the following:

```
ThisPt          Pixel   {,10}

ThisPt          Pixel   {{},10}

ThisPt          Pixel   {{1,2,3}, 10}

ThisPt          Pixel   {{1,,1}, 12}
```

The first and second examples above use the default values for the Pt field (x=0, y=0, z=0) and set the Color field to 10. Note the use of braces to surround the initial values for the point type in the second, third, and fourth examples. The third example above initializes the x, y, and z fields of the Pt field to one, two, and three, respectively. The last example initializes the x and z fields to one and lets the y field take on the initial value specified by the Point structure (zero).

Accessing Pixel fields is very easy. Like a high level language you use a single period to reference the Pt field and a second period to access the x, y, and z fields of point:

14. Note, of course, that the initial values for the x, y , and z fields need not all be zero. You could have initialized the fields to 1, 2, and 3 just as easily.

```
                mov     ax, ThisPt.Pt.X
                 .
                 .
                 .
                mov     ThisPt.Pt.Y, 0
                 .
                 .
                 .
                mov     ThisPt.Pt.Z, di
                 .
                 .
                 .
                mov     ThisPt.Color, EAX
```

You can also declare *arrays* as structure fields. The following structure creates a data type capable of representing an object with eight points (e.g., a cube):

```
Object8         struct
Pts             point   8 dup (?)
Color           dword   0
Object8         ends
```

This structure allocates storage for eight different points. Accessing an element of the Pts array requires that you know the size of an object of type point (remember, you must multiply the index into the array by the size of one element, six in this particular case). Suppose, for example, that you have a variable CUBE of type Object8. You could access elements of the Pts array as follows:

```
; CUBE.Pts[i].X := 0;

                mov     ax, 6
                mul     i           ;6 bytes per element.
                mov     si, ax
                mov     CUBE.Pts[si].X, 0
```

The one unfortunate aspect of all this is that you must know the size of each element of the Pts array. Fortunately, MASM provides an operator that will compute the size of an array element (in bytes) for you, more on that later.

5.6.5 Pointers to Structures

During execution, your program may refer to structure objects directly or indirectly using a pointer. When you use a pointer to access fields of a structure, you must load one of the 80x86's pointer registers (si, di, bx, or bp on processors less than the 80386) with the offset and es, ds, ss, or cs[15] with the segment of the desired structure. Suppose you have the following variable declarations (assuming the Object8 structure from the previous section):

```
Cube            Object8 {}
CubePtr         dword   Cube
```

CubePtr contains the address of (i.e., it is a pointer to) the Cube object. To access the Color field of the Cube object, you could use an instruction like mov eax,Cube.Color. When accessing a field via a pointer you need to load the address of the object into a segment:pointer register pair, such as es:bx. The instruction les bx,CubePtr will do the trick. After doing so, you can access fields of the Cube object using the disp+bx addressing mode. The only problem is "How do you specify which field to access?" Consider briefly, the following *incorrect* code:

```
                les     bx, CubePtr
                mov     eax, es:[bx].Color
```

15. Add FS or GS to this list for the 80386 and later.

There is one major problem with the code above. Since field names are local to a structure and it's possible to reuse a field name in two or more structures, how does MASM determine which offset Color represents? When accessing structure members directly (.e.g., mov eax,Cube.Color) there is no ambiguity since Cube has a specific type that the assembler can check. es:bx, on the other hand, can point at *anything*. In particular, it can point at any structure that contains a Color field. So the assembler cannot, on its own, decide which offset to use for the Color symbol.

MASM resolves this ambiguity by requiring that you explicitly supply a type in this case. Probably the easiest way to do this is to specify the structure name as a *pseudo-field*:

```
les     bx, CubePtr
mov     eax, es:[bx].Object8.Color
```

By specifying the structure name, MASM knows which offset value to use for the Color symbol[16].

5.7 Sample Programs

The following short sample programs demonstrate many of the concepts appearing in this chapter.

5.7.1 Simple Variable Declarations

```
; Sample variable declarations
; This sample file demonstrates how to declare and access some simple
; variables in an assembly language program.
;
; Randall Hyde
;
;
; Note: global variable declarations should go in the "dseg" segment:

dseg            segment  para public 'data'

; Some simple variable declarations:

character       byte     ?           ;"?" means uninitialized.
UnsignedIntVar  word     ?
DblUnsignedVar  dword    ?

;You can use the typedef statement to declare more meaningful type names:

integer         typedef  sword
char            typedef  byte
FarPtr          typedef  dword

; Sample variable declarations using the above types:

J               integer  ?
c1              char     ?
PtrVar          FarPtr   ?


; You can tell MASM & DOS to initialize a variable when DOS loads the
; program into memory by specifying the initial value in the operand
```

16. Users of MASM 5.1 and other assemblers should keep in mind that field names are *not* local to the structure. Instead, they must all be unique within a source file. As a result, such programs do not require the structure name in the "dot path" for a particular field. Keep this in mind when converting older code to MASM 6.x.

```
; field of the variable's declaration:

K               integer  4
c2              char     'A'
PtrVar2         FarPtr   L            ;Initializes PtrVar2 with the
                                     ; address of L.


; You can also set aside more than one byte, word, or double word of
; storage using these directives.  If you place several values in the
; operand field, separated by commas, the assembler will emit one byte,
; word, or dword for each operand:

L               integer  0, 1, 2, 3
c3              char     'A', 0dh, 0ah, 0
PtrTbl          FarPtr   J, K, L

; The BYTE directive lets you specify a string of characters byte enclosing
; the string in quotes or apostrophes.  The directive emits one byte of data
; for every character in the string (not including the quotes or apostrophes
; that delimit the string):

string          byte     "Hello world",0dh,0ah,0


dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg;These statements are provided by
                mov      ds, ax       ; shell.asm to initialize the
                mov      es, ax       ; segment register.


; Some simple instructions that demonstrate how to access memory:

                lea      bx, L        ;Point bx at first word in L.
                mov      ax, [bx];Fetch word at L.
                add      ax, 2[bx];Add in word at L+2 (the "1").
                add      ax, 4[bx];Add in word at L+4 (the "2").
                add      ax, 6[bx];Add in word at L+6 (the "3").
                mul      K            ;Compute (0+1+2+3)*123.
                mov      J, ax        ;Save away result in J.

                les      bx, PtrVar2;Loads es:di with address of L.
                mov      di, K        ;Loads 4 into di
                mov      ax, es:[bx][di];Fetch value of L+4.


; Examples of some byte accesses:

                mov      c1, ' '      ;Put a space into the c1 var.
                mov      al, c2       ;c3 := c2
                mov      c3, al
```

```
Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.2 Using Pointer Variables

```
; Using Pointer Variables in an Assembly Language Program
;
; This short sample program demonstrates the use of pointers in
; an assembly language program.
;
; Randall Hyde

dseg            segment  para public 'data'


; Some variables we will access indirectly (using pointers):

J               word     0, 0, 0, 0
K               word     1, 2, 3, 4
L               word     5, 6, 7, 8

; Near pointers are 16-bits wide and hold an offset into the current data
; segment (dseg in this program).  Far pointers are 32-bits wide and hold
; a complete segment:offset address.  The following type definitions let
; us easily create near and far pointers

nWrdPtr         typedef  near ptr word
fWrdPtr         typedef  far ptr word


; Now for the actual pointer variables:

Ptr1            nWrdPtr  ?
Ptr2            nWrdPtr  K          ;Initialize with K's address.
Ptr3            fWrdPtr  L          ;Initialize with L's segmented adrs.

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg   ;These statements are provided by
                mov      ds, ax     ; shell.asm to initialize the
                mov      es, ax     ; segment register.
```

```
; Initialize Ptr1 (a near pointer) with the address of the J variable.

                lea     ax, J
                mov     Ptr1, ax

; Add the four words in variables J, K, and L together using pointers to
; these variables:

                mov     bx, Ptr1    ;Get near ptr to J's 1st word.
                mov     si, Ptr2    ;Get near ptr to K's 1st word.
                les     di, Ptr3    ;Get far ptr to L's 1st word.



                mov     ax, ds:[si] ;Get data at K+0.
                add     ax, es:[di] ;Add in data at L+0.
                mov     ds:[bx], ax ;Store result to J+0.

                add     bx, 2       ;Move to J+2.
                add     si, 2       ;Move to K+2.
                add     di, 2       ;Move to L+2.



                mov     ax, ds:[si] ;Get data at K+2.
                add     ax, es:[di] ;Add in data at L+2.
                mov     ds:[bx], ax ;Store result to J+2.

                add     bx, 2       ;Move to J+4.
                add     si, 2       ;Move to K+4.
                add     di, 2       ;Move to L+4.



                mov     ax, ds:[si] ;Get data at K+4.
                add     ax, es:[di] ;Add in data at L+4.
                mov     ds:[bx], ax ;Store result to J+4.

                add     bx, 2       ;Move to J+6.
                add     si, 2       ;Move to K+6.
                add     di, 2       ;Move to L+6.



                mov     ax, ds:[si] ;Get data at K+6.
                add     ax, es:[di] ;Add in data at L+6.
                mov     ds:[bx], ax ;Store result to J+6.



Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.3 Single Dimension Array Access

```
; Sample variable declarations
; This sample file demonstrates how to declare and access some single
; dimension array variables in an assembly language program.
;
; Randall Hyde


                .386                            ;Need to use some 80386
                option  segment:use16           ; addressing modes.

dseg            segment para public 'data'

J               word    ?
K               word    ?
L               word    ?
M               word    ?

JD              dword   0
KD              dword   1
LD              dword   2
MD              dword   3

; Some simple uninitialized array declarations:

ByteAry         byte    4 dup (?)
WordAry         word    4 dup (?)
DwordAry        dword   4 dup (?)
RealAry         real8   4 dup (?)


; Some arrays with initialized values:

BArray          byte    0, 1, 2, 3
WArray          word    0, 1, 2, 3
DWArray         dword   0, 1, 2, 3
RArray          real8   0.0, 1.0, 2.0, 3.0


; An array of pointers:

PtrArray        dword   ByteAry, WordAry, DwordAry, RealAry

dseg            ends

; The following program demonstrates how to access each of the above
; variables.

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.


; Initialize the index variables.  Note that these variables provide
; logical indices into the arrays.  Don't forget that we've got to
; multiply these values by the element size when accessing elements of
; an array.
```

```
                mov     J, 0
                mov     K, 1
                mov     L, 2
                mov     M, 3

; The following code shows how to access elements of the arrays using
; simple 80x86 addressing modes:

                mov     bx, J       ;AL := ByteAry[J]
                mov     al, ByteAry[bx]

                mov     bx, K       ;AX := WordAry[K]
                add     bx, bx      ;Index*2 since this is a word array.
                mov     ax, WordAry[bx]

                mov     bx, L       ;EAX := DwordAry[L]
                add     bx, bx      ;Index*4 since this is a double
                add     bx, bx      ; word array.
                mov     eax, DwordAry[bx]

                mov     bx, M       ;BX := address(RealAry[M])
                add     bx, bx      ;Index*8 since this is a quad
                add     bx, bx      ; word array.
                add     bx, bx
                lea     bx, RealAry[bx];Base address + index*8.

; If you have an 80386 or later CPU, you can use the 386's scaled indexed
; addressing modes to simplify array access.

                mov     ebx, JD
                mov     al, ByteAry[ebx]

                mov     ebx, KD
                mov     ax, WordAry[ebx*2]

                mov     ebx, LD
                mov     eax, DwordAry[ebx*4]

                mov     ebx, MD
                lea     bx, RealAry[ebx*8]

Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.4 Multidimensional Array Access

```
; Multidimensional Array declaration and access
;
```

```
; Randall Hyde


                .386                            ;Need these two statements to
                option  segment:use16           ; use the 80386 register set.


dseg            segment para public 'data'


; Indices we will use for the arrays.

J               word    1
K               word    2
L               word    3

; Some two-dimensional arrays.
; Note how this code uses the "dup" operator to suggest the size
; of each dimension.

B2Ary           byte    3 dup (4 dup (?))
W2Ary           word    4 dup (3 dup (?))
D2Ary           dword   2 dup (6 dup (?))



; 2D arrays with initialization.
; Note the use of data layout to suggest the sizes of each array.

B2Ary2          byte    0, 1, 2, 3
                byte    4, 5, 6, 7
                byte    8, 9, 10, 11

W2Ary2          word    0,  1,  2
                word    3,  4,  5
                word    6,  7,  8
                word    9, 10, 11

D2Ary2          dword   0,  1,  2,  3,  4,  5
                dword   6,  7,  8,  9, 10, 11

; A sample three dimensional array.

W3Ary           word    2 dup (3 dup (4 dup (?)))

dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.

; AL := B2Ary2[j,k]

                mov     bx, J       ;index := (j*4+k)
                add     bx, bx      ;j*2
                add     bx, bx      ;j*4
                add     bx, K       ;j*4+k
                mov     al, B2Ary2[bx]
```

```
; AX := W2Ary2[j,k]

                mov     ax, J       ;index := (j*3 + k)*2
                mov     bx, 3
                mul     bx          ;(j*3)-- This destroys DX!
                add     ax, k       ;(j*3+k)
                add     ax, ax      ;(j*3+k)*2
                mov     bx, ax
                mov     ax, W2Ary2[bx]


; EAX := D2Ary[i,j]

                mov     ax, J       ;index := (j*6 + k)*4
                mov     bx, 6
                mul     bx          ;DX:AX := j*6, ignore overflow in DX.
                add     ax, k       ;j*6 + k
                add     ax, ax      ;(j*6 + k)*2
                add     ax, ax      ;(j*6 + k)*4
                mov     bx, ax
                mov     eax, D2Ary[bx]


; Sample access of a three dimensional array.
;
; AX := W3Ary[J,K,L]

                mov     ax, J       ;index := ((j*3 + k)*4 + l)*2
                mov     bx, 3
                mul     bx          ;j*3
                add     ax, K       ;j*3 + k
                add     ax, ax      ;(j*3 + k)*2
                add     ax, ax      ;(j*3 + k)*4
                add     ax, l       ;(j*3 + k)*4 + l
                add     ax, ax      ;((j*3 + k)*4 + l)*2
                mov     bx, ax
                mov     ax, W3Ary[bx]


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.5 Simple Structure Access

```
; Sample Structure Definitions and Accesses.
;
; Randall Hyde
```

```
dseg            segment  para public 'data'


; The following structure holds the bit values for an 80x86 mod-reg-r/m byte.

mode            struct
modbits         byte     ?
reg             byte     ?
rm              byte     ?
mode            ends


Instr1Adrs      mode     {};All fields uninitialized.
Instr2Adrs      mode     {}


; Some structures with initialized fields.

axbx            mode     {11b, 000b, 000b}      ;"ax, ax" adrs mode.
axdisp          mode     {00b, 000b, 110b}      ;"ax, disp" adrs mode.
cxdispbxsi      mode     {01b, 001b, 000b}      ;"cx, disp8[bx][si]" mode.


; Near pointers to some structures:

sPtr1           word     axdisp
sPtr2           word     Instr2Adrs

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg     ;These statements are provided by
                mov      ds, ax       ; shell.asm to initialize the
                mov      es, ax       ; segment register.


; To access fields of a structure variable directly, just use the "."
; operator like you would in Pascal or C:

                mov      al, axbx.modbits
                mov      Instr1Adrs.modbits, al

                mov      al, axbx.reg
                mov      Instr1Adrs.reg, al

                mov      al, axbx.rm
                mov      Instr1Adrs.rm, al


; When accessing elements of a structure indirectly (that is, using a
; pointer) you must specify the structure type name as the first
; "field" so MASM doesn't get confused:

                mov      si, sPtr1
                mov      di, sPtr2

                mov      al, ds:[si].mode.modbits
                mov      ds:[di].mode.modbits, al
```

```
                mov     al, ds:[si].mode.reg
                mov     ds:[di].mode.reg, al

                mov     al, ds:[si].mode.rm
                mov     ds:[di].mode.rm, al


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.7.6 Arrays of Structures

```
; Arrays of Structures
;
; Randall Hyde


dseg            segment para public 'data'


; A structure that defines an (x,y) coordinate.
; Note that the Point data type requires four bytes.

Point           struct
X               word    ?
Y               word    ?
Point           ends



; An uninitialized point:

Pt1             Point   {}

; An initialized point:

Pt2             Point   {12,45}


; A one-dimensional array of uninitialized points:

PtAry1          Point   16 dup ({})  ;Note the "{}" inside the parens.


; A one-dimensional array of points, all initialized to the origin.

PtAry1i         Point   16 dup ({0,0})
```

```
; A two-dimensional array of points:

PtAry2          Point    4 dup (4 dup ({}))


; A three-dimensional array of points, all initialized to the origin.

PtAry3          Point    2 dup (3 dup (4 dup ({0,0})))



; A one-dimensional array of points, all initialized to different values:

iPtAry          Point    {0,0}, {1,2}, {3,4}, {5,6}


; Some indices for the arrays:

J               word     1
K               word     2
L               word     3

dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg     ;These statements are provided by
                mov      ds, ax       ; shell.asm to initialize the
                mov      es, ax       ; segment register.

; PtAry1[J] := iPtAry[J]

                mov      bx, J        ;Index := J*4 since there are four
                add      bx, bx       ; bytes per array element (each
                add      bx, bx       ; element contains two words).

                mov      ax, iPtAry[bx].X
                mov      PtAry1[bx].X, ax

                mov      ax, iPtAry[bx].Y
                mov      PtAry1[bx].Y, ax

; CX := PtAry2[K,L].X;  DX := PtAry2[K,L].Y

                mov      bx, K        ;Index := (K*4 + J)*4
                add      bx, bx       ;K*2
                add      bx, bx       ;K*4
                add      bx, J        ;K*4 + J
                add      bx, bx       ;(K*4 + J)*2
                add      bx, bx       ;(K*4 + J)*4

                mov      cx, PtAry2[bx].X
                mov      dx, PtAry2[bx].Y

; PtAry3[j,k,l].X := 0
```

```
                mov     ax, j       ;Index := ((j*3 +k)*4 + l)*4
                mov     bx, 3
                mul     bx          ;j*3
                add     ax, k       ;j*3 + k
                add     ax, ax      ;(j*3 + k)*2
                add     ax, ax      ;(j*3 + k)*4
                add     ax, l       ;(j*3 + k)*4 + l
                add     ax, ax      ;((j*3 + k)*4 + l)*2
                add     ax, ax      ;((j*3 + k)*4 + l)*4
                mov     bx, ax
                mov     PtAry3[bx].X, 0

Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.7 Structures and Arrays as Fields of Another Structure

```
; Structures Containing Structures as fields
; Structures Containing Arrays as fields
;
; Randall Hyde


dseg            segment  para public 'data'

Point           struct
X               word     ?
Y               word     ?
Point           ends

; We can define a rectangle with only two points.
; The color field contains an eight-bit color value.
; Note: the size of a Rect is 9 bytes.

Rect            struct
UpperLeft       Point    {}
LowerRight      Point    {}
Color           byte     ?
Rect            ends

; Pentagons have five points, so use an array of points to
; define the pentagon.  Of course, we also need the color
; field.
; Note: the size of a pentagon is 21 bytes.

Pent            struct
Color           byte     ?
Pts             Point    5 dup ({})
Pent            ends
```

```
; Okay, here are some variable declarations:

Rect1           Rect    {}
Rect2           Rect    {{0,0}, {1,1}, 1}

Pentagon1       Pent    {}
Pentagons       Pent    {}, {}, {}, {}

Index           word    2

dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.

; Rect1.UpperLeft.X := Rect2.UpperLeft.X

                mov     ax, Rect2.Upperleft.X
                mov     Rect1.Upperleft.X, ax

; Pentagon1 := Pentagons[Index]

                mov     ax, Index;Need Index*21
                mov     bx, 21
                mul     bx
                mov     bx, ax

; Copy the first point:

                mov     ax, Pentagons[bx].Pts[0].X
                mov     Pentagon1.Pts[0].X, ax

                mov     ax, Pentagons[bx].Pts[0].Y
                mov     Pentagon1.Pts[0].Y, ax

; Copy the second point:

                mov     ax, Pentagons[bx].Pts[2].X
                mov     Pentagon1.Pts[4].X, ax

                mov     ax, Pentagons[bx].Pts[2].Y
                mov     Pentagon1.Pts[4].Y, ax

; Copy the third point:

                mov     ax, Pentagons[bx].Pts[4].X
                mov     Pentagon1.Pts[8].X, ax

                mov     ax, Pentagons[bx].Pts[4].Y
                mov     Pentagon1.Pts[8].Y, ax

; Copy the fourth point:

                mov     ax, Pentagons[bx].Pts[6].X
                mov     Pentagon1.Pts[12].X, ax
```

```
                mov     ax, Pentagons[bx].Pts[6].Y
                mov     Pentagon1.Pts[12].Y, ax

; Copy the fifth point:

                mov     ax, Pentagons[bx].Pts[8].X
                mov     Pentagon1.Pts[16].X, ax

                mov     ax, Pentagons[bx].Pts[8].Y
                mov     Pentagon1.Pts[16].Y, ax

; Copy the Color:

                mov     al, Pentagons[bx].Color
                mov     Pentagon1.Color, al


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

5.7.8 Pointers to Structures and Arrays of Structures

```
; Pointers to structures
; Pointers to arrays of structures
;
; Randall Hyde


                .386                            ;Need these two statements so
                option   segment:use16          ; we can use 80386 registers

dseg            segment  para public 'data'

; Sample structure.
; Note: size is seven bytes.

Sample          struct
b               byte     ?
w               word     ?
d               dword    ?
Sample          ends


; Some variable declarations:

OneSample       Sample   {}
SampleAry       Sample   16 dup ({})

; Pointers to the above
```

```
OnePtr          word    OneSample   ;A near pointer.
AryPtr          dword   SampleAry


; Index into the array:

Index           word    8

dseg            ends




; The following program demonstrates how to access each of the above
; variables.

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg    ;These statements are provided by
                mov     ds, ax      ; shell.asm to initialize the
                mov     es, ax      ; segment register.

; AryPtr^[Index] := OnePtr^


                mov     si, OnePtr  ;Get pointer to OneSample
                les     bx, AryPtr  ;Get pointer to array of samples
                mov     ax, Index   ;Need index*7
                mov     di, 7
                mul     di
                mov     di, ax

                mov     al, ds:[si].Sample.b
                mov     es:[bx][di].Sample.b, al

                mov     ax, ds:[si].Sample.w
                mov     es:[bx][di].Sample.w, ax

                mov     eax, ds:[si].Sample.d
                mov     es:[bx][di].Sample.d, eax


Quit:           mov     ah, 4ch     ;Magic number for DOS
                int     21h         ; to tell this program to quit.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.8 Laboratory Exercises

In these laboratory exercises you will learn how to step through a program using CodeView and observe the results. Knowing how to trace through a program is an important skill to posses. There is no better way to learn assembly language than to single step through a program and observe the actions of each instruction. Even if you already know assembly language, tracing through a program with a debugger like CodeView is one of the best ways to verify that your program is working properly.

In these lab exercises you will assemble the sample program provided in the previous section. Then you will run the assembled program under CodeView and step through each instruction in the program. **For your lab report:** you will include a listing of each program and describe the operation of each statement including data loaded into any affected registers or values stored away into memory.

The following paragraphs describe one experimental run – stepping through the pgm5_1.asm program. Your lab report should contain similar information for all eight sample programs.

To assemble your programs, use the ML command with the /Zi option. For example, to assemble the first sample program you would use the following DOS command:

```
ml /Zi pgm5_1.asm
```

This command produces the pgm5_1.exe file that contains CodeView debugging information. You can load this program into the CodeView debugger using the following command:

```
cv pgm5_1
```

Once you are inside CodeView, you can single step through the program by repeatedly pressing the F8 key. Each time you press the F8 key, CodeView executes a single instruction in the program.

To better observe the results while stepping through your program, you should open the register window. If it is not open already, you can open it by pressing the F2 key. As the instructions you execute modify the registers, you can observe the changes.

All the sample programs begin with a three-instruction sequence that initializes the DS and ES registers; pressing the F8 key three times steps over these instructions and (on one system) loads the AX, ES, and DS registers with the value 1927h (this value will change on different systems).

Single stepping over the `lea bx, L` instruction loads the value 0015h into bx. Single stepping over the group of instructions following the lea produces the following results:

```
mov     ax, [bx]            ;AX = 0
add     ax, 2[bx]           ;AX = 1
add     ax, 4[bx]           ;AX = 3
add     ax, 6[bx]           ;AX = 6
mul     K                   ;AX = 18 (hex)
mov     J, ax               ;J is now equal to 18h.
```

Comments on the above instructions: this code loads bx with the base address of array L and then proceeds to compute the sum of L[i], i=0..3 (0+1+2+3). It then multiples this sum by K (4) and stores the result into J. Note that you can use the "dw J" command in the command window to display J's current value (the "J" must be upper case because CodeView is case sensitive).

```
les     bx, PtrVar2         ;BX = 0015, ES = 1927
mov     di, K               ;DI = 4
mov     ax, es:[bx][di]     ;AX = 2
```

Comments on the above code: The les instruction loads es:bx with the pointer variable PtrVar2. This variable contains the address of the L variable. Then this code loads di with the value of K and completes by loading the second element of L into ax.

```
mov     c1, ' '
mov     al, c2
mov     c3, al
```

These three instructions simply store a space into byte variable c1 (verify with a "da c1" command in the command window) and they copy the value in c2 ("A") into the AL register and the c3 variable (verified with "da c3" command).

For your lab report: assemble and step through pgm5_2.asm, pgm5_3.asm, pgm5_4.asm, pgm5_5.asm, pgm5_6.asm, pgm5_7.asm, and pgm5_8.asm. Describe the results in a fashion similar to the above.

5.9 Programming Projects

1) The PC's video display is a *memory mapped I/O device*. That is, the display adapter maps each character on the text display to a word in memory. The display is an 80x25 array of words declared as follows:

```
display:array[0..24,0..79] of word;
```

Display[0,0] corresponds to the upper left hand corner of the screen, display[0,79] is the upper right hand corner, display[24,0] is the lower left hand corner, and display[24,79] is the lower right hand corner of the display.

The L.O. byte of each word holds the ASCII code of the character to appear on the screen. The H.O. byte of each word contains the *attribute* byte (see "The PC Video Display" on page 1247 for more details on the attribute byte). The base address of this array is B000:0 for monochrome displays and B800:0 for color displays.

The Chapter Five subdirectory contains a file named PGM5_1.ASM. This file is a skeletal program that manipulates the PC's video display. This program, when complete, writes a series of period to the screen and then it writes a series of blues spaces to the screen. It contains a main program that uses several instructions you probably haven't seen yet. These instructions essentially execute a for loop as follows:

```
for i:= 0 to 79 do
          for j := 0 to 24 do
                    putscreen(i,j,value);
```

Inside this program you will find some comments that instruct you to supply the code that stores the value in AX to location display[i,j]. Modify this program as described in its comments and test the result.

For this project, you need to declare two word variables, I and J, in the data segment. Then you will need to modify the "PutScreen" procedure. Inside this procedure, as directed by the comments in the file, you will need to compute the index into the screen array and then store the value in the ax register to location es:[bx+0] (assuming you've computed the index into bx). Note that es:[0] is the base address of the video display in this procedure. Check your code carefully before attempting to run it. If your code malfunctions, it may crash the system and you will have to reboot. This program, if operating properly, will fill the screen with periods and wait until you press a key. Then it will fill the screen with blue spaces. You should probably execute the DOS "CLS" (clear screen) command after this program executes properly. Note that there is a working version of this program named p5_1.exe in the Chapter Five directory. You can run this program to check out it's operation if you are having problems.

2) The Chapter Five subdirectory contains a file named PGM5_2.ASM. This file is a program (except for two short subroutines) that generates mazes and solves them on the screen. This program requires that you complete two subroutines: MazeAdrs and ScrnAdrs. These two procedures appear at the beginning of the file; you should ignore the remainder

of this program. When the program calls the MazeAdrs function, it passes an X coordinate in the dx register and a Y-coordinate in the cx register. You need to compute the index into an 27x82 array defined as follows:

```
maze:array[0..26, 0..81] of word;
```

Return the index in the ax register. *Do not access the maze array; the calling code will do that for you.*

The ScrnAdrs function is almost identical to the MazeAdrs function except it computes an index into a 25x80 array rather than a 27x82 array. As with MazeAdrs, the X-coordinate will be in the dx register and the Y-coordinate will be in the cx register.

Complete these two functions, assemble the program, and run it. Be sure to check your work over carefully. If you make any mistakes you will probably crash the system.

3) Create a program with a single dimension array of structures. Place at least four fields (your choice) in the structure. Write a code segment to access element "i" ("i" being a word variable) in the array.

4) Write a program which copies the data from a 3x3 array and stores the data into a second 3x3 array. For the first 3x3 array, store the data in row major order. For the second 3x3 array, store the data in column major order. Use nine sequences of instructions which fetch the word at location (i,j) (i=0..2, j=0..2).

5) Rewrite the code sequence above just using MOV instructions. Read and write the array locations directly, do not perform the array address computations.

5.10 Summary

This chapter presents an 80x86-centric view of memory organization and data structures. This certainly isn't a complete course on data structures. This chapter discussed the primitive and simple composite data types and how to declare and use them in your program. Lots of additional information on the declaration and use of simple data types appears later in this text.

One of the main goals of this chapter is to describe how to declare and use *variables* in an assembly language program. In an assembly language program you can easily create byte, word, double word, and other types of variables. Such scalar data types support boolean, character, integer, real, and other single data types found in typical high level languages. See:

- "Declaring Variables in an Assembly Language Program" on page 196
- "Declaring and using BYTE Variables" on page 198
- "Declaring and using WORD Variables" on page 200
- "Declaring and using DWORD Variables" on page 201
- "Declaring and using FWORD, QWORD, and TBYTE Variables" on page 202
- "Declaring Floating Point Variables with REAL4, REAL8, and REAL10" on page 202

For those who don't like using variable type names like `byte`, `word`, etc., MASM lets you create your own type names. You want to call them *Integers* rather than *Words*? No problem, you can define your own type names use the `typedef` statement. See:

- "Creating Your Own Type Names with TYPEDEF" on page 203

Another important data type is the *pointer*. Pointers are nothing more than memory addresses stored in variables (word or double word variables). The 80x86 CPUs support two types of pointers – near and far pointers. In real mode, near pointers are 16 bits long and contain the offset into a known segment (typically the data segment). Far pointers are 32 bits long and contain a full segment:offset logical address. Remember that you must use one of the register indirect or indexed addressing modes to access the data referenced by a pointer. For those who want to create their own pointer types (rather than simply

using `word` and `dword` to declare near and far pointers), the `typedef` instruction lets you create named pointer types. See:

- “Pointer Data Types” on page 203

A *composite data type* is one made up from other data types. Examples of composite data types abound, but two of the more popular composite data types are arrays and structures (records). An array is a group of variables, all the same type. A program selects an element of an array using an integer index into that array. Structures, on the other hand, may contain fields whose types are different. In a program, you select the desired field by supplying a field name with the *dot operator*. See:

- “Arrays” on page 206
- “Multidimensional Arrays” on page 210
- “Structures” on page 218
- “Arrays of Structures and Arrays/Structures as Structure Fields” on page 220
- “Pointers to Structures” on page 221

5.11 Questions

1) In what segment (8086) would you normally place your variables?

2) Which segment in the SHELL.ASM file normally corresponds to the segment containing your variables?

3) Describe how to declare byte variables. Give several examples. What would you normally use byte variables for in a program?

4) Describe how to declare word variables. Give several examples. Describe what you would use them for in a program.

5) Repeat question 21 for double word variables.

6) Explain the purpose of the TYPEDEF statement. Give several examples of its use.

7) What is a pointer variable?

8) What is the difference between a *near* and a *far* pointer?

9) How do you access the object pointed at by a far pointer. Give an example using 8086 instructions.

10) What is a composite data type?

11) How do you declare arrays in assembly language? Give the code for the following arrays:

a) A two dimensional 4x4 array of bytes b) An array containing 128 double words

c) An array containing 16 words d) A 4x5x6 three dimensional array of words

12) Describe how you would access a single element of each of the above arrays. Provide the necessary formulae and 8086 code to access said element (assume variable I is the index into single dimension arrays, I & J provide the index into two dimension arrays, and I, J, & K provide the index into the three dimensional array). Assume row major ordering, where appropriate.

13) Provide the 80386 code, using the scaled indexing modes, to access the elements of the above arrays.

14) Explain the difference between row major and column major array ordering.

15) Suppose you have a two-dimensional array whose values you want to initialize as follows:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Provide the variable declaration to accomplish this. Note: Do not use 8086 machine instructions to initialize the array. Initialize the array in your data segment.

```
Date=           Record
                        Month:integer;
                        Day:integer;
                        Year:integer;
                end;

Time=           Record
                        Hours:integer;
                        Minutes:integer;
                        Seconds:integer;
                end;

VideoTape =             record
                          Title:string[25];
                          ReleaseDate:Date;
                          Price:Real; (* Assume four byte reals *)
```

```
                                    Length: Time;
                                    Rating:char;
                                  end;

TapeLibrary : array [0..127] of VideoTape; (*This is a variable!*)
```

17) Suppose ES:BX points at an object of type VideoTape. What is the instruction that properly loads the `Rating` field into AL?

The 80x86 Instruction Set Chapter Six

Until now, there has been little discussion of the instructions available on the 80x86 microprocessor. This chapter rectifies this situation. Note that this chapter is mainly for *reference*. It explains what each instruction does, it does not explain how to combine these instructions to form complete assembly language programs. The rest of this book will explain how to do that.

6.0 Chapter Overview

This chapter discusses the 80x86 real mode instruction set. Like any programming language, there are going to be several instructions you use all the time, some you use occasionally, and some you will rarely, if ever, use. This chapter organizes its presentation by instruction class rather than importance. Since beginning assembly language programmers do not have to learn the entire instruction set in order to write meaningful assembly language programs, you will probably not have to learn how every instruction operates. The following list describes the instructions this chapter discusses. A "•" symbol marks the important instructions in each group. If you learn only these instructions, you will probably be able to write any assembly language program you want. There are many additional instructions, especially on the 80386 and later processors. These additional instructions make assembly language programming easier, but you do not need to know them to begin writing programs.

80x86 instructions can be (roughly) divided into eight different classes:

1) Data movement instructions
   - mov, lea, les , push, pop, pushf, popf
2) Conversions
   - cbw, cwd, xlat
3) Arithmetic instructions
   - add, inc sub, dec, cmp, neg, mul, imul, div, idiv
4) Logical, shift, rotate, and bit instructions
   - and, or, xor, not, shl, shr, rcl, rcr
5) I/O instructions
   - in, out
6) String instructions
   - movs, stos, lods
7) Program flow control instructions
   - jmp, call, ret, conditional jumps
8) Miscellaneous instructions.
   - clc, stc, cmc

The following sections describe all the instructions in these groups and how they operate.

At one time a text such as this one would recommend against using the extended 80386 instruction set. After all, programs that use such instructions will not run properly on 80286 and earlier processors. Using these additional instructions could limit the number of machines your code would run on. However, the 80386 processor is on the verge of disappearing as this text is being written. You can safely assume that most systems will contain an 80386sx or later processor. This text often uses the 80386 instruction set in various example programs. Keep in mind, though, that this is only for convenience. There is no program that appears in this text that could not be recoded using only 8088 assembly language instructions.

A word of advice, particularly to those who learn only the instructions noted above: as you read about the 80x86 instruction set you will discover that the individual 80x86 instructions are not very complex and have simple semantics. However, as you approach

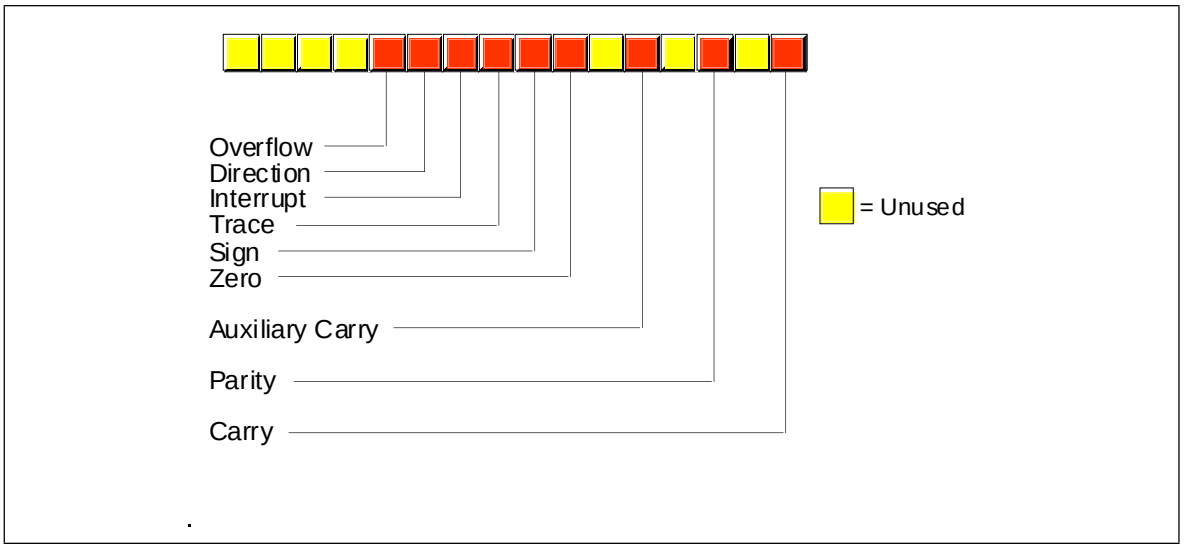


Figure 6.1 80x86 Flags Register

the end of this chapter, you may discover that you haven't got a clue how to put these simple instructions together to form a complex program. Fear not, this is a common problem. Later chapters will describe how to form complex programs from these simple instructions.

One quick note: this chapter lists many instructions as "available only on the 80286 and later processors." In fact, many of these instructions were available on the 80186 microprocessor as well. Since few PC systems employ the 80186 microprocessor, this text ignores that CPU. However, to keep the record straight...

6.1 The Processor Status Register (Flags)

The flags register maintains the current operating mode of the CPU and some instruction state information. Figure 6.1 shows the layout of the flags register.

The carry, parity, zero, sign, and overflow flags are special because you can test their status (zero or one) with the set*cc* and conditional jump instructions (see "The "Set on Condition" Instructions" on page 281 and "The Conditional Jump Instructions" on page 296). The 80x86 uses these bits, the *condition codes*, to make decisions during program execution.

Various arithmetic, logical, and miscellaneous instructions affect the *overflow flag*. After an arithmetic operation, this flag contains a one if the result does not fit in the signed destination operand. For example, if you attempt to add the 16 bit signed numbers 7FFFh and 0001h the result is too large so the CPU sets the overflow flag. If the result of the arithmetic operation does not produce a signed overflow, then the CPU clears this flag.

Since the logical operations generally apply to unsigned values, the 80x86 logical instructions simply clear the overflow flag. Other 80x86 instructions leave the overflow flag containing an arbitrary value.

The 80x86 string instructions use the *direction flag*. When the direction flag is clear, the 80x86 processes string elements from low addresses to high addresses; when set, the CPU processes strings in the opposite direction. See "String Instructions" on page 284 for additional details.

The *interrupt enable/disable flag* controls the 80x86's ability to respond to external events known as interrupt requests. Some programs contain certain instruction sequences that the CPU must not interrupt. The interrupt enable / disable flag turns interrupts on or off to guarantee that the CPU does not interrupt those critical sections of code.

The *trace flag* enables or disables the 80x86 trace mode. Debuggers (such as CodeView) use this bit to enable or disable the single step / trace operation. When set, the CPU interrupts each instruction and passes control to the debugger software, allowing the debugger to *single step* through the application. If the trace bit is clear, then the 80x86 executes instructions without the interruption. The 80x86 CPUs do not provide any instructions that directly manipulate this flag. To set or clear the trace flag, you must:

- Push the flags onto the 80x86 stack,
- Pop the value into another register,
- Tweak the trace flag value,
- Push the result onto the stack, and then
- Pop the flags off the stack.

If the result of some computation is negative, the 80x86 sets the *sign flag*. You can test this flag after an arithmetic operation to check for a negative result. Remember, a value is negative if its H.O. bit is one. Therefore, operations on unsigned values will set the sign flag if the result has a one in the H.O. position.

Various instructions set the *zero flag* when they generate a zero result. You'll often use this flag to see if two values are equal (e.g., after subtracting two numbers, they are equal if the result is zero). This flag is also useful after various logical operations to see if a specific bit in a register or memory location contains zero or one.

The *auxiliary carry flag* supports special binary coded decimal (BCD) operations. Since most programs don't deal with BCD numbers, you'll rarely use this flag and even then you'll not access it directly. The 80x86 CPUs do not provide any instructions that let you directly test, set, or clear this flag. Only the add, adc, sub, sbb, mul, imul, div, idiv, and BCD instructions manipulate this flag.

The *parity flag* is set according to the parity of the L.O. eight bits of any data operation. If an operation produces an even number of one bits, the CPU sets this flag. It clears this flag if the operation yields an odd number of one bits. This flag is useful in certain data communications programs, however, Intel provided it mainly to provide some compatibility with the older 8080 µP.

The *carry flag* has several purposes. First, it denotes an unsigned overflow (much like the overflow flag detects a signed overflow). You will also use it during multiprecision arithmetic and logical operations. Certain bit test, set, clear, and invert instructions on the 80386 directly affect this flag. Finally, since you can easily clear, set, invert, and test it, it is useful for various boolean operations. The carry flag has many purposes and knowing when to use it, and for what purpose, can confuse beginning assembly language programmers. Fortunately, for any given instruction, the meaning of the carry flag is clear.

The use of these flags will become readily apparent in the coming sections and chapters. This section is mainly a formal introduction to the individual flags in the register rather than an attempt to explain the exact function of each flag. For more details on the operation of each flag, keep reading...

6.2 Instruction Encodings

The 80x86 uses a binary encoding for each machine operation. While it is important to have a general understanding of how the 80x86 encodes instructions, it is not important that you memorize the encodings for all the instructions in the instruction set. If you were to write an assembler or disassembler (debugger), you would definitely need to know the exact encodings. For general assembly language programming, however, you won't need to know the exact encodings.

However, as you become more experienced with assembly language you will probably want to study the encodings of the 80x86 instruction set. Certainly you should be aware of such terms as *opcode, mod-reg-r/m byte, displacement value*, and so on. Although you do not need to memorize the parameters for each instruction, it is always a good idea to know the lengths and cycle times for instructions you use regularly since this will help

you write better programs. Chapter Three and Chapter Four provided a detailed look at instruction encodings for various instructions (80x86 and x86); such a discussion was important because you do need to understand how the CPU encodes and executes instructions. This chapter does not deal with such details. This chapter presents a higher level view of each instruction and assumes that you don't care how the machine treats bits in memory. For those few times that you will need to know the binary encoding for a particular instruction, a complete listing of the instruction encodings appears in Appendix D.

6.3 Data Movement Instructions

The data movement instructions copy values from one location to another. These instructions include mov, xchg, lds, lea, les, lfs, lgs, lss, push, pusha, pushad, pushf, pushfd, pop, popa, popad, popf, popfd, lahf, and sahf.

6.3.1 The MOV Instruction

The mov instruction takes several different forms:

```
mov     reg, reg[1]
mov     mem, reg
mov     reg, mem
mov     mem, immediate data
mov     reg, immediate data
mov     ax/al, mem
mov     mem, ax/al
mov     segreg, mem16
mov     segreg, reg16
mov     mem16, segreg
mov     reg16, segreg
```

The last chapter discussed the mov instruction in detail, only a few minor comments are worthwhile here. First, there are variations of the mov instruction that are faster and shorter than other mov instructions that do the same job. For example, both the mov ax, mem and mov reg, mem instructions can load the ax register from a memory location. On all processors the first version is shorter. On the earlier members of the 80x86 family, it is faster as well.

There are two very important details to note about the mov instruction. First, there is no memory to memory move operation. The mod-reg-r/m addressing mode byte (see Chapter Four) allows two register operands or a single register and a single memory operand. There is no form of the mov instruction that allows you to encode *two* memory addresses into the same instruction. Second, you cannot move immediate data into a segment register. The only instructions that move data into or out of a segment register have mod-reg-r/m bytes associated with them; there is no format that moves an immediate value into a segment register. Two common errors beginning programmers make are attempting a memory to memory move and trying to load a segment register with a constant.

The operands to the mov instruction may be bytes, words, or double words[2]. Both operands must be the same size or MASM will generate an error while assembling your program. This applies to memory operands and register operands. If you declare a variable, B, using byte and attempt to load this variable into the ax register, MASM will complain about a type conflict.

The CPU extends immediate data to the size of the destination operand (unless it is too big to fit in the destination operand, which is an error). Note that you *can* move an

1. This chapter uses "reg", by itself, to denote any eight bit, sixteen bit, or (on the 80386 and later) 32 bit general purpose CPU register (AL/AX/EAX, BL/BX/EBX, SI/ESI, etc.)
2. Double word operands are valid only on 80386 and later processors.

immediate value into a memory location. The same rules concerning size apply. However, MASM cannot determine the size of certain memory operands. For example, does the instruction `mov [bx], 0` store an eight bit, sixteen bit, or thirty-two bit value? MASM cannot tell, so it reports an error. This problem does *not* exist when you move an immediate value into a variable you've declared in your program. For example, if you've declared `B` as a byte variable, MASM knows to store an eight bit zero into `B` for the instruction `mov B, 0`. Only those memory operands involving pointers with no variable operands suffer from this problem. The solution is to explicitly tell MASM whether the operand is a byte, word, or double word. You can accomplish this with the following instruction forms:

```
mov     byte ptr [bx], 0
mov     word ptr [bx], 0
mov     dword ptr [bx], 0          (3)
```

(3) Available only on 80386 and later processors

For more details on the *type* ptr operator, see Chapter Eight.

Moves to and from segment registers are always 16 bits; the mod-reg-r/m operand must be 16 bits or MASM will generate an error. Since you cannot load a constant directly into a segment register, a common solution is to load the constant into an 80x86 general purpose register and then copy it to the segment register. For example, the following two instruction sequence loads the `es` register with the value 40h:

```
mov     ax, 40h
mov     es, ax
```

Note that almost any general purpose register would suffice. Here, `ax` was chosen arbitrarily.

The `mov` instructions do not affect any flags. In particular, the 80x86 preserves the flag values across the execution of a `mov` instruction.

6.3.2 The XCHG Instruction

The `xchg` (exchange) instruction swaps two values. The general form is

```
xchg    operand1, operand2
```

There are four specific forms of this instruction on the 80x86:

```
xchg    reg, mem
xchg    reg, reg
xchg    ax, reg16
xchg    eax, reg32    (3)
```

(3) Available only on 80386 and later processors

The first two general forms require two or more bytes for the opcode and mod-reg-r/m bytes (a displacement, if necessary, requires additional bytes). The third and fourth forms are special forms of the second that exchange data in the (e)ax register with another 16 or 32 bit register. The 16 bit form uses a single byte opcode that is shorter than the other two forms that use a one byte opcode and a mod-reg-r/m byte.

Already you should note a pattern developing: the 80x86 family often provides shorter and faster versions of instructions that use the ax register. Therefore, you should try to arrange your computations so that they use the (e)ax register as much as possible. The `xchg` instruction is a perfect example, the form that exchanges 16 bit registers is only one byte long.

Note that the order of the `xchg`'s operands does not matter. That is, you could enter `xchg mem, reg` and get the same result as `xchg reg, mem`. Most modern assemblers will automatically emit the opcode for the shorter `xchg ax, reg` instruction if you specify `xchg reg, ax`.

Both operands must be the same size. On pre-80386 processors the operands may be eight or sixteen bits. On 80386 and later processors the operands may be 32 bits long as well.

The xchg instruction does not modify any flags.

6.3.3 The LDS, LES, LFS, LGS, and LSS Instructions

The lds, les, lfs, lgs, and lss instructions let you load a 16 bit general purpose register and segment register pair with a single instruction. On the 80286 and earlier, the lds and les instructions are the only instructions that directly process values larger than 32 bits. The general form is

```
LxS     dest, source
```

These instructions take the specific forms:

```
lds     reg16, mem32
les     reg16, mem32
lfs     reg16, mem32            (3)
lgs     reg16, mem32            (3)
lss     reg16, mem32            (3)
```

(3) Available only on 80386 and later processors

Reg_{16} is any general purpose 16 bit register and mem_{32} is a double word memory location (declared with the dword statement).

These instructions will load the 32 bit double word at the address specified by mem_{32} into reg_{16} and the ds, es, fs, gs, or ss registers. They load the general purpose register from the L.O. word of the memory operand and the segment register from the H.O. word. The following algorithms describe the exact operation:

```
lds reg16, mem32:
                reg16 := [mem32]
                ds := [mem32 + 2]
les reg16, mem32:
                reg16 := [mem32]
                es := [mem32 + 2]
lfs reg16, mem32:
                reg16 := [mem32]
                fs := [mem32 + 2]
lgs reg16, mem32:
                reg16 := [mem32]
                gs := [mem32 + 2]
lss reg16, mem32:
                reg16 := [mem32]
                ss := [mem32 + 2]
```

Since the LxS instructions load the 80x86's segment registers, you must not use these instructions for arbitrary purposes. Use them to set up (far) pointers to certain data objects as discussed in Chapter Four. Any other use may cause problems with your code if you attempt to port it to Windows, OS/2 or UNIX.

Keep in mind that these instructions load the four bytes at a given memory location into the register pair; they do *not* load the address of a variable into the register pair (i.e., this instruction does not have an immediate mode). To learn how to load the address of a variable into a register pair, see Chapter Eight.

The LxS instructions do not affect any of the 80x86's flag bits.

6.3.4 The LEA Instruction

The lea (Load Effective Address) instruction is another instruction used to prepare pointer values. The lea instruction takes the form:

```
lea     dest, source
```

The specific forms on the 80x86 are

```
lea     reg16, mem
lea     reg32, mem              (3)
```

(3) Available only on 80386 and later processors.

It loads the specified 16 or 32 bit general purpose register with the *effective address* of the specified memory location. The effective address is the final memory address obtained after all addressing mode computations. For example, lea ax, ds:[1234h] loads the ax register with the address of memory location 1234h; here it just loads the ax register with the value 1234h. If you think about it for a moment, this isn't a very exciting operation. After all, the mov ax, immediate_data instruction can do this. So why bother with the lea instruction at all? Well, there are many other forms of a memory operand besides displacement-only operands. Consider the following lea instructions:

```
lea     ax, [bx]
lea     bx, 3[bx]
lea     ax, 3[bx]
lea     bx, 4[bp+si]
lea     ax, -123[di]
```

The lea ax, [bx] instruction copies the address of the expression [bx] into the ax register. Since the effective address is the value in the bx register, this instruction copies bx's value into the ax register. Again, this instruction isn't very interesting because mov can do the same thing, even faster.

The lea bx,3[bx] instruction copies the effective address of 3[bx] into the bx register. Since this effective address is equal to the current value of bx plus three, this lea instruction effectively adds three to the bx register. There is an add instruction that will let you add three to the bx register, so again, the lea instruction is superfluous for this purpose.

The third lea instruction above shows where lea really begins to shine. lea ax, 3[bx] copies the address of the memory location 3[bx] into the ax register; i.e., it adds three with the value in the bx register and moves the sum into ax. This is an excellent example of how you can use the lea instruction to do a mov operation and an addition with a single instruction.

The final two instructions above, lea bx,4[bp+si] and lea ax,-123[di] provide additional examples of lea instructions that are more efficient than their mov/add counterparts.

On the 80386 and later processors, you can use the scaled indexed addressing modes to multiply by two, four, or eight as well as add registers and displacements together. Intel *strongly* suggests the use of the lea instruction since it is much faster than a sequence of instructions computing the same result.

The (real) purpose of lea is to load a register with a memory address. For example, lea bx, 128[bp+di] sets up bx with the address of the byte referenced by 128[BP+DI]. As it turns out, an instruction of the form mov al,[bx] runs faster than an instruction of the form mov al,128[bp+di]. If this instruction executes several times, it is probably more efficient to load the effective address of 128[bp+di] into the bx register and use the [bx] addressing mode. This is a common optimization in high performance programs.

The lea instruction does not affect any of the 80x86's flag bits.

6.3.5 The PUSH and POP Instructions

The 80x86 push and pop instructions manipulate data on the 80x86's hardware stack. There are 19 varieties of the push and pop instructions[3], they are

3. Plus some synonyms on top of these 19.

```
push    reg16
pop     reg16
push    reg32                   (3)
pop     reg32                   (3)
push    segreg
pop     segreg                  (except CS)
push    memory
pop     memory
push    immediate_data          (2)
pusha                           (2)
popa                            (2)
pushad                          (3)
popad                           (3)
pushf
popf
pushfd                          (3)
popfd                           (3)
enter   imm, imm                (2)
leave                           (2)

(2)- Available only on 80286 and later processors.
(3)- Available only on 80386 and later processors.
```

The first two instructions push and pop a 16 bit general purpose register. This is a compact (one byte) version designed specifically for registers. Note that there is a second form that provides a mod-reg-r/m byte that could push registers as well; most assemblers only use that form for pushing the value of a memory location.

The second pair of instructions push or pop an 80386 32 bit general purpose register. This is really nothing more than the push register instruction described in the previous paragraph with a size prefix byte.

The third pair of push/pop instructions let you push or pop an 80x86 segment register. Note that the instructions that push fs and gs are longer than those that push cs, ds, es, and ss, see Appendix D for the exact details. You can only push the cs register (popping the cs register would create some interesting program flow control problems).

The fourth pair of push/pop instructions allow you to push or pop the contents of a memory location. On the 80286 and earlier, this must be a 16 bit value. For memory operations without an explicit type (e.g., [bx]) you must either use the pushw mnemonic or explicitly state the size using an instruction like push word ptr [bx]. On the 80386 and later you can push and pop 16 or 32 bit values[4]. You can use dword memory operands, you can use the pushd mnemonic, or you can use the dword ptr operator to force 32 bit operation. Examples:

```
push    DblWordVar
push    dword ptr [bx]
pushd   dword
```

The pusha and popa instructions (available on the 80286 and later) push and pop *all* the 80x86 16 bit general purpose registers. Pusha pushes the registers in the following order: ax, cx, dx, bx, sp, bp, si, and then di. Popa pops these registers in the reverse order. Pushad and Popad (available on the 80386 and later) do the same thing on the 80386's 32 bit register set. Note that these "push all" and "pop all" instructions do *not* push or pop the flags or segment registers.

The pushf and popf instructions allow you to push/pop the processor status register (the flags). Note that these two instructions provide a mechanism to modify the 80x86's trace flag. See the description of this process earlier in this chapter. Of course, you can set and clear the other flags in this fashion as well. However, most of the other flags you'll want to modify (specifically, the condition codes) provide specific instructions or other simple sequences for this purpose.

Enter and leave push/pop the bp register and allocate storage for local variables on the stack. You will see more on these instructions in a later chapter. This chapter does not con-

4. You can use the PUSHW and PUSHD mnemonics to denote 16 or 32 bit constant sizes.

sider them since they are not particularly useful outside the context of procedure entry and exit.

"So what do these instructions do?" you're probably asking by now. The push instructions move data onto the 80x86 hardware stack and the pop instructions move data from the stack to memory or to a register. The following is an algorithmic description of each instruction:

push instructions (16 bits):

```
        SP := SP - 2
        [SS:SP] := 16 bit operand (store result at location SS:SP.)
pop instructions (16 bits):
        16-bit operand := [SS:SP]
        SP := SP + 2
push instructions (32 bits):
        SP := SP - 4
        [SS:SP] := 32 bit operand
pop instructions (32 bits):
        32 bit operand := [SS:SP]
        SP := SP + 4
```

You can treat the pusha/pushad and popa/popad instructions as equivalent to the corresponding sequence of 16 or 32 bit push/pop operations (e.g., push ax, push cx, push dx, push bx, etc.).

Notice three things about the 80x86 hardware stack. First, it is always in the stack segment (wherever ss points). Second, the stack grows down in memory. That is, as you push values onto the stack the CPU stores them into successively lower memory locations. Finally, the 80x86 hardware stack pointer (ss:sp) always contains the address of the value on the top of the stack (the last value pushed on the stack).

You can use the 80x86 hardware stack for temporarily saving registers and variables, passing parameters to a procedure, allocating storage for local variables, and other uses. The push and pop instructions are extremely valuable for manipulating these items on the stack. You'll get a chance to see how to use them later in this text.

Most of the push and pop instructions do not affect any of the flags in the 80x86 processor status register. The popf/popfd instructions, by their very nature, can modify all the flag bits in the 80x86 processor status register (flags register). Pushf and pushfd push the flags onto the stack, but they do not change any flags while doing so.

All pushes and pops are 16 or 32 bit operations. There is no (easy) way to push a single eight bit value onto the stack. To push an eight bit value you would need to load it into the H.O. byte of a 16 bit register, push that register, and then add one to the stack pointer. On all processors except the 8088, this would slow future stack access since sp now contains an odd address, misaligning any further pushes and pops. Therefore, most programs push or pop 16 bits, even when dealing with eight bit values.

Although it is relatively safe to push an eight bit memory variable, be careful when popping the stack to an eight bit memory location. Pushing an eight bit variable with push word ptr ByteVar pushes two bytes, the byte in the variable ByteVar and the byte immediately following it. Your code can simply ignore the extra byte this instruction pushes onto the stack. Popping such values is not quite so straight forward. Generally, it doesn't hurt if you push these two bytes. However, it can be a disaster if you pop a value and wipe out the following byte in memory. There are only two solutions to this problem. First, you could pop the 16 bit value into a register like ax and then store the L.O. byte of that register into the byte variable. The second solution is to reserve an extra byte of padding after the byte variable to hold the whole word you will pop. Most programs use the former approach.

6.3.6 The LAHF and SAHF Instructions

The lahf (load ah from flags) and sahf (store ah into flags) instructions are archaic instructions included in the 80x86's instruction set to help improve compatibility with Intel's older 8080 µP chip. As such, these instructions have very little use in modern day 80x86 programs. The lahf instruction does not affect any of the flag bits. The sahf instruction, by its very nature, modifies the S, Z, A, P, and C bits in the processor status register. These instructions do not require any operands and you use them in the following manner:

```
sahf
lahf
```

Sahf only affects the L.O. eight bits of the flags register. Likewise, lahf only loads the L.O. eight bits of the flags register into the AH register. These instructions do not deal with the overflow, direction, interrupt disable, or trace flags. The fact that these instructions do not deal with the overflow flag is an important limitation.

Sahf has one major use. When using a floating point processor (8087, 80287, 80387, 80486, Pentium, etc.) you can use the sahf instruction to copy the floating point status register flags into the 80x86's flag register. You'll see this use in the chapter on floating point arithmetic (see "Floating Point Arithmetic" on page 771).

6.4 Conversions

The 80x86 instruction set provides several conversion instructions. They include movzx, movsx, cbw, cwd, cwde, cdq, bswap, and xlat. Most of these instructions sign or zero extend values, the last two convert between storage formats and translate values via a lookup table. These instructions take the general form:

```
movzx    dest, src    ;Dest must be twice the size of src.
movsx    dest, src    ;Dest must be twice the size of src.
cbw
cwd
cwde
cdq
bswap    reg32
xlat                  ;Special form allows an operand.
```

6.4.1 The MOVZX, MOVSX, CBW, CWD, CWDE, and CDQ Instructions

These instructions zero and sign extend values. The cbw and cwd instructions are available on all 80x86 processors. The movzx, movsx, cwde, and cdq instructions are available only on 80386 and later processors.

The cbw (convert byte to word) instruction sign extends the eight bit value in al to ax. That is, it copies bit seven of AL throughout bits 8-15 of ax. This instruction is especially important before executing an eight bit division (as you'll see in the section "Arithmetic Instructions" on page 255). This instruction requires no operands and you use it as follows:

```
cbw
```

The cwd (convert word to double word) instruction sign extends the 16 bit value in ax to 32 bits and places the result in dx:ax. It copies bit 15 of ax throughout the bits in dx. It is available on all 80x86 processors which explains why it doesn't sign extend the value into eax. Like the cbw instruction, this instruction is very important for division operations. Cwd requires no operands and you use it as follows

```
cwd
```

The cwde instruction sign extends the 16 bit value in ax to 32 bits and places the result in eax by copying bit 15 of ax throughout bits 16..31 of eax. This instruction is available only on the 80386 and later processors. As with cbw and cwd the instruction has no operands and you use it as follows:

```
cwde
```

The cdq instruction sign extends the 32 bit value in eax to 64 bits and places the result in edx:eax by copying bit 31 of eax throughout bits 0..31 of edx. This instruction is available only on the 80386 and later. You would normally use this instruction before a long division operation. As with cbw, cwd, and cwde the instruction has no operands and you use it as follows:

```
cdq
```

If you want to sign extend an eight bit value to 32 or 64 bits using these instructions, you could use sequences like the following:

```
; Sign extend al to dx:ax

        cbw
        cwd

; Sign extend al to eax

        cbw
        cwde

; Sign extend al to edx:eax

        cbw
        cwde
        cdq
```

You can also use the movsx for sign extensions from eight to sixteen or thirty-two bits.

The movsx instruction is a generalized form of the cbw, cwd, and cwde instructions. It will sign extend an eight bit value to a sixteen or thirty-two bits, or sign extend a sixteen bit value to a thirty-two bits. This instruction uses a mod-reg-r/m byte to specify the two operands. The allowable forms for this instruction are

```
movsx   reg16, mem8
movsx   reg16, reg8
movsx   reg32, mem8
movsx   reg32, reg8
movsx   reg32, mem16
movsx   reg32, reg16
```

Note that anything you can do with the cbw and cwde instructions, you can do with a movsx instruction:

```
movsx   ax, al      ;CBW
movsx   eax, ax     ;CWDE
movsx   eax, al     ;CBW followed by CWDE
```

However, the cbw and cwde instructions are shorter and sometimes faster. This instruction is available only on the 80386 and later processors. Note that there are not direct movsx equivalents for the cwd and cdq instructions.

The movzx instruction works just like the movsx instruction, except it extends unsigned values via zero extension rather than signed values through sign extension. The syntax is the same as for the movsx instructions except, of course, you use the movzx mnemonic rather than movsx.

Note that if you want to zero extend an eight bit register to 16 bits (e.g., al to ax) a simple mov instruction is faster and shorter than movzx. For example,

```
mov     bh, 0
```

is faster and shorter than

```
movzx   bx, bl
```

Of course, if you move the data to a different 16 bit register (e.g., movzx bx, al) the movzx instruction is better.

Like the movsx instruction, the movzx instruction is available only on 80386 and later processors. The sign and zero extension instructions do not affect any flags.

6.4.2 The BSWAP Instruction

The bswap instruction, available only on 80486 (yes, 486) and later processors, converts between 32 bit *little endian* and *big endian* values. This instruction accepts only a single 32 bit register operand. It swaps the first byte with the fourth and the second byte with the third. The syntax for the instruction is

```
bswap     reg32
```

where reg_{32} is an 80486 32 bit general purpose register.

The Intel processor families use a memory organization known as *little endian byte organization*. In little endian byte organization, the L.O. byte of a multi-byte sequence appears at the lowest address in memory. For example, bits zero through seven of a 32 bit value appear at the lowest address; bits eight through fifteen appear at the second address in memory; bits 16 through 23 appear in the third byte, and bits 24 through 31 appear in the fourth byte.

Another popular memory organization is *big endian*. In the big endian scheme, bits twenty-four through thirty-one appear in the first (lowest) address, bits sixteen through twenty-three appear in the second byte, bits eight through fifteen appear in the third byte, and bits zero through seven appear in the fourth byte. CPUs such as the Motorola 68000 family used by Apple in their Macintosh computer and many RISC chips employ the big endian scheme.

Normally, you wouldn't care about byte organization in memory since programs written for an Intel processor in assembly language do not run on a 68000 processor. However, it is very common to exchange data between machines with different byte organizations. Unfortunately, 16 and 32 bit values on big endian machines do not produce correct results when you use them on little endian machines. This is where the bswap instruction comes in. It lets you easily convert 32 bit big endian values to 32 bit little endian values.

One interesting use of the bswap instruction is to provide access to a second set of 16 bit general purpose registers. If you are using only 16 bit registers in your code, you can double the number of available registers by using the bswap instruction to exchange the data in a 16 bit register with the H.O. word of a thirty-two bit register. For example, you can keep two 16 bit values in eax and move the appropriate value into ax as follows:

```
< Some computations that leave a result in AX >
          bswap     eax
< Some additional computations involving AX >
          bswap     eax
< Some computations involving the original value in AX >
          bswap     eax
< Computations involving the 2nd copy of AX from above >
```

You can use this technique on the 80486 to obtain two copies of ax, bx, cx, dx, si, di, and bp. You must exercise extreme caution if you use this technique with the sp register.

Note: to convert 16 bit big endian values to 16 bit little endian values just use the 80x86 xchg instruction. For example, if ax contains a 16 bit big endian value, you can convert it to a 16 bit little endian value (or vice versa) using:

```
xchg      al, ah
```

The bswap instruction does not affect any flags in the 80x86 flags register.

6.4.3 The XLAT Instruction

The xlat instruction translates the value in the al register based on a lookup table in memory. It does the following:

```
temp := al+bx
al := ds:[temp]
```

that is, bx points at a table in the current data segment. Xlat replaces the value in al with the byte at the offset originally in al. If al contains four, xlat replaces the value in al with the fifth item (offset four) within the table pointed at by ds:bx. The xlat instruction takes the form:

```
xlat
```

Typically it has no operand. You can specify one but the assembler virtually ignores it. The only purpose for specifying an operand is so you can provide a segment override prefix:

```
xlat      es:Table
```

This tells the assembler to emit an es: segment prefix byte before the instruction. You must still load bx with the address of Table; the form above does not provide the address of Table to the instruction. Only the segment override prefix in the operand is significant.

The xlat instruction does not affect the 80x86's flags register.

6.5 Arithmetic Instructions

The 80x86 provides many arithmetic operations: addition, subtraction, negation, multiplication, division/modulo (remainder), and comparing two values. The instructions that handle these operations are add, adc, sub, sbb, mul, imul, div, idiv, cmp, neg, inc, dec, xadd, cmpxchg, and some miscellaneous conversion instructions: aaa, aad, aam, aas, daa, and das. The following sections describe these instructions in detail.

The generic forms for these instructions are

```
add    dest, src              dest := dest + src
adc    dest, src              dest := dest + src + C
SUB    dest, src              dest := dest - src
sbb    dest, src              dest := dest - src - C
mul    src                    acc := acc * src
imul   src                    acc := acc * src
imul   dest, src1, imm_src    dest := src1 * imm_src
imul   dest, imm_src          dest := dest * imm_src
imul   dest, src              dest := dest * src
div    src                    acc := xacc /-mod src
idiv   src                    acc := xacc /-mod src
cmp    dest, src              dest - src (and set flags)
neg    dest                   dest := - dest
inc    dest                   dest := dest + 1
dec    dest                   dest := dest - 1
xadd   dest, src              (see text)
cmpxchg operand1, operand2    (see text)
cmpxchg8ax, operand           (see text)
aaa                           (see text)
aad                           (see text)
aam                           (see text)
aas                           (see text)
daa                           (see text)
das                           (see text)
```

6.5.1 The Addition Instructions: ADD, ADC, INC, XADD, AAA, and DAA

These instructions take the forms:

```
add     reg, reg
add     reg, mem
add     mem, reg
add     reg, immediate data
add     mem, immediate data
add     eax/ax/al, immediate data

adc forms are identical to ADD.

inc     reg
inc     mem
inc     reg16
xadd    mem, reg
xadd    reg, reg
aaa
daa
```

Note that the aaa and daa instructions use the implied addressing mode and allow no operands.

6.5.1.1 The ADD and ADC Instructions

The syntax of add and adc (add with carry) is similar to mov. Like mov, there are special forms for the ax/eax register that are more efficient. Unlike mov, you cannot add a value to a segment register with these instructions.

The add instruction adds the contents of the source operand to the destination operand. For example, add ax, bx adds bx to ax leaving the sum in the ax register. Add computes dest:=dest+source while adc computes dest:=dest+source+C where C represents the value in the carry flag. Therefore, if the carry flag is clear before execution, adc behaves exactly like the add instruction.

Both instructions affect the flags identically. They set the flags as follows:

- The overflow flag denotes a signed arithmetic overflow.
- The carry flag denotes an unsigned arithmetic overflow.
- The sign flag denotes a negative result (i.e., the H.O. bit of the result is one).
- The zero flag is set if the result of the addition is zero.
- The auxiliary carry flag contains one if a BCD overflow out of the L.O. nibble occurs.
- The parity flag is set or cleared depending on the parity of the L.O. eight bits of the result. If there are an even number of one bits in the result, the ADD instructions will set the parity flag to one (to denote *even parity*). If there are an odd number of one bits in the result, the ADD instructions clear the parity flag (to denote *odd parity*).

The add and adc instructions do not affect any other flags.

The add and adc instructions allow eight, sixteen, and (on the 80386 and later) thirty-two bit operands. Both source and destination operands must be the same size. See Chapter Nine if you want to add operands whose size is different.

Since there are no memory to memory additions, you must load memory operands into registers if you want to add two variables together. The following code examples demonstrate possible forms for the add instruction:

```
; J:= K + M

                mov     ax, K
                add     ax, M
                mov     J, ax
```

If you want to add several values together, you can easily compute the sum in a single register:

```
; J := K + M + N + P

                mov     ax, K
                add     ax, M
                add     ax, N
                add     ax, P
                mov     J, ax
```

If you want to reduce the number of hazards on an 80486 or Pentium processor, you can use code like the following:

```
                mov     bx, K
                mov     ax, M
                add     bx, N
                add     ax, P
                add     ax, bx
                mov     J, ax
```

One thing that beginning assembly language programmers often forget is that you can add a register to a memory location. Sometimes beginning programmers even believe that both operands have to be in registers, completely forgetting the lessons from Chapter Four. The 80x86 is a CISC processor that allows you to use memory addressing modes with various instructions like add. It is often more efficient to take advantages of the 80x86's memory addressing capabilities

```
; J := K + J

                mov     ax, K           ;This works because addition is
                add     J, ax           ; commutative!

; Often, beginners will code the above as one of the following two sequences.
; This is unnecessary!

                mov     ax, J           ;Really BAD way to compute
                mov     bx, K           ; J := J + K.
                add     ax, bx
                mov     J, ax

                mov     ax, J           ;Better, but still not a good way to
                add     ax, K           ; compute J := J + K
                mov     J, ax
```

Of course, if you want to add a constant to a memory location, you only need a single instruction. The 80x86 lets you directly add a constant to memory:

```
; J := J + 2

                add     J, 2
```

There are special forms of the add and adc instructions that add an immediate constant to the al, ax, or eax register. These forms are shorter than the standard add reg, immediate instruction. Other instructions also provide shorter forms when using these registers; therefore, you should try to keep computations in the accumulator registers (al, ax, and eax) as much as possible.

```
                add     bl, 2           ;Three bytes long
                add     al, 2           ;Two bytes long
                add     bx, 2           ;Four bytes long
                add     ax, 2           ;Three bytes long
                etc.
```

Another optimization concerns the use of small signed constants with the add and adc instructions. If a value is in the range -128,,+127, the add and adc instructions will sign extend an eight bit immediate constant to the necessary destination size (eight, sixteen, or thirty-two bits). Therefore, you should try to use small constants, if possible, with the add and adc instructions.

6.5.1.2 The INC Instruction

The inc (increment) instruction adds one to its operand. Except for the carry flag, inc sets the flags the same way as add operand, 1 would.

Note that there are two forms of inc for 16 or 32 bit registers. They are the inc reg and inc reg_{16} instructions. The inc reg and inc mem instructions are the same. This instruction consists of an opcode byte followed by a mod-reg-r/m byte (see Appendix D for details). The inc reg_{16} instruction has a single byte opcode. Therefore, it is shorter and usually faster.

The inc operand may be an eight bit, sixteen bit, or (on the 80386 and later) thirty-two bit register or memory location.

The inc instruction is more compact and often faster than the comparable add reg, 1 or add mem, 1 instruction. Indeed, the inc reg_{16} instruction is one byte long, so it turns out that *two* such instructions are shorter than the comparable add reg, 1 instruction; however, the two increment instructions will run slower on most modern members of the 80x86 family.

The inc instruction is very important because adding one to a register is a very common operation. Incrementing loop control variables or indices into an array is a very common operation, perfect for the inc instruction. The fact that inc does not affect the carry flag is very important. This allows you to increment array indices without affecting the result of a multiprecision arithmetic operation (see “Arithmetic and Logical Operations” on page 459 for more details about multiprecision arithmetic).

6.5.1.3 The XADD Instruction

Xadd (Exchange and Add) is another 80486 (and later) instruction. It does not appear on the 80386 and earlier processors. This instruction adds the source operand to the destination operand and stores the sum in the destination operand. However, just before storing the sum, it copies the original value of the destination operand into the source operand. The following algorithm describes this operation:

```
xadd dest, source

temp := dest
dest := dest + source
source := temp
```

The xadd sets the flags just as the add instruction would. The xadd instruction allows eight, sixteen, and thirty-two bit operands. Both source and destination operands must be the same size.

6.5.1.4 The AAA and DAA Instructions

The aaa (ASCII adjust after addition) and daa (decimal adjust for addition) instructions support BCD arithmetic. Beyond this chapter, this text will not cover BCD or ASCII arithmetic since it is mainly for controller applications, not general purpose programming applications. BCD values are decimal integer coded in binary form with one decimal digit (0..9) per nibble. ASCII (numeric) values contain a single decimal digit per byte, the H.O. nibble of the byte should contain zero.

The aaa and daa instructions modify the result of a binary addition to correct it for ASCII or decimal arithmetic. For example, to add two BCD values, you would add them as though they were binary numbers and then execute the daa instruction afterwards to correct the results. Likewise, you can use the aaa instruction to adjust the result of an ASCII addition after executing an add instruction. Please note that these two instructions assume that the add operands were proper decimal or ASCII values. If you add binary

(non-decimal or non-ASCII) values together and try to adjust them with these instructions, you will not produce correct results.

The choice of the name "ASCII arithmetic" is unfortunate, since these values are not true ASCII characters. A name like "unpacked BCD" would be more appropriate. However, Intel uses the name ASCII, so this text will do so as well to avoid confusion. However, you will often hear the term "unpacked BCD" to describe this data type.

Aaa (which you generally execute after an add, adc, or xadd instruction) checks the value in al for BCD overflow. It works according to the following basic algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC[5] =1) ) then
                if (8088 or 8086)[6] then
                        al := al + 6
                else
                        ax := ax + 6
                endif
                ah := ah + 1
                AuxC := 1                       ;Set auxilliary carry
                Carry := 1                      ; and carry flags.
else
                AuxC := 0                       ;Clear auxilliary carry
                Carry := 0                      ; and carry flags.
endif
al := al and 0Fh
```

The aaa instruction is mainly useful for adding strings of digits where there is exactly one decimal digit per byte in a string of numbers. This text will not deal with BCD or ASCII numeric strings, so you can safely ignore this instruction for now. Of course, you can use the aaa instruction any time you need to use the algorithm above, but that would probably be a rare situation.

The daa instruction functions like aaa except it handles packed BCD (binary code decimal) values rather than the one digit per byte unpacked values aaa handles. As for aaa, daa's main purpose is to add strings of BCD digits (with two digits per byte). The algorithm for daa is

```
if ( (AL and 0Fh) > 9 or (AuxC = 1)) then
                al := al + 6
                AuxC := 1                       ;Set Auxilliary carry.
endif
if ( (al > 9Fh) or (Carry = 1)) then
                al := al + 60h
                Carry := 1;                     ;Set carry flag.
endif
```

6.5.2 The Subtraction Instructions: SUB, SBB, DEC, AAS, and DAS

The sub (subtract), sbb (subtract with borrow), dec (decrement), aas (ASCII adjust for subtraction), and das (decimal adjust for subtraction) instructions work as you expect. Their syntax is very similar to that of the add instructions:

```
sub     reg, reg
sub     reg, mem
sub     mem, reg
sub     reg, immediate data
sub     mem, immediate data
sub     eax/ax/al, immediate data
```

5. AuxC denotes the *auxiliary carry* flag in the flags register.
6. The 8086/8088 work differently from the later processors, but for all valid operands all 80x86 processors produce correct results.

```
                sbb forms are identical to sub.

                dec     reg
                dec     mem
                dec     reg16
                aas
                das
```

The sub instruction computes the value dest := dest - src. The sbb instruction computes dest := dest - src - C. Note that subtraction is not commutative. If you want to compute the result for dest := src - dest you will need to use several instructions, assuming you need to preserve the source operand).

One last subject worth discussing is how the sub instruction affects the 80x86 flags register[7]. The sub, sbb, and dec instructions affect the flags as follows:

- They set the zero flag if the result is zero. This occurs only if the operands are equal for sub and sbb. The dec instruction sets the zero flag only when it decrements the value one.
- These instructions set the sign flag if the result is negative.
- These instructions set the overflow flag if signed overflow/underflow occurs.
- They set the auxiliary carry flag as necessary for BCD/ASCII arithmetic.
- They set the parity flag according to the number of one bits appearing in the result value.
- The sub and sbb instructions set the carry flag if an unsigned overflow occurs. Note that the dec instruction does not affect the carry flag.

The aas instruction, like its aaa counterpart, lets you operate on strings of ASCII numbers with one decimal digit (in the range 0..9) per byte. You would use this instruction after a sub or sbb instruction on the ASCII value. This instruction uses the following algorithm:

```
if ( (al and 0Fh) > 9 or AuxC = 1) then
                al := al - 6
                ah := ah - 1
                AuxC := 1               ;Set auxilliary carry
                Carry := 1              ; and carry flags.
else
                AuxC := 0               ;Clear Auxilliary carry
                Carry := 0              ; and carry flags.
endif
al := al and 0Fh
```

The das instruction handles the same operation for BCD values, it uses the following algorithm:

```
if ( (al and 0Fh) > 9 or (AuxC = 1)) then
                al := al -6
                AuxC = 1
endif
if (al > 9Fh or Carry = 1) then
                al := al - 60h
                Carry := 1              ;Set the Carry flag.
endif
```

Since subtraction is not commutative, you cannot use the sub instruction as freely as the add instruction. The following examples demonstrate some of the problems you may encounter.

```
; J := K - J
                mov     ax, K           ;This is a nice try, but it computes
                sub     J, ax           ; J := J - K, subtraction isn't
                                        ; commutative!
```

7. The SBB instruction affects the flags in a similar fashion, just don't forget that SBB computes dest-source-C.

```
                mov     ax, K       ;Correct solution.
                sub     ax, J
                mov     J, ax

; J := J - (K + M) -- Don't forget this is equivalent to J := J - K - M

                mov     ax, K       ;Computes AX := K + M
                add     ax, M
                sub     J, ax       ;Computes J := J - (K + M)

                mov     ax, J       ;Another solution, though less
                sub     ax, K       ;Efficient
                sub     ax, M
                mov     J, ax
```

Note that the sub and sbb instructions, like add and adc, provide short forms to subtract a constant from an accumulator register (al, ax, or eax). For this reason, you should try to keep arithmetic operations in the accumulator registers as much as possible. The sub and sbb instructions also provide a shorter form when subtracting constants in the range -128..+127 from a memory location or register. The instruction will automatically sign extend an eight bit signed value to the necessary size before the subtraction occurs. See Appendix D for the details.

In practice, there really isn't a need for an instruction that subtracts a constant from a register or memory location – adding a negative value achieves the same result. Nevertheless, Intel provides a subtract immediate instruction.

After the execution of a sub instruction, the condition code bits (carry, sign, overflow, and zero) in the flags register contain values you can test to see if one of sub's operands is equal, not equal, less than, less than or equal, greater than, or greater than or equal to the other operand. See the cmp instruction for more details.

6.5.3 The CMP Instruction

The cmp (compare) instruction is identical to the sub instruction with one crucial difference – it does not store the difference back into the destination operand. The syntax for the cmp instruction is very similar to sub, the generic form is

```
                cmp     dest, src
```

The specific forms are

```
                cmp     reg, reg
                cmp     reg, mem
                cmp     mem, reg
                cmp     reg, immediate data
                cmp     mem, immediate data
                cmp     eax/ax/al, immediate data
```

The cmp instruction updates the 80x86's flags according to the result of the subtraction operation (dest - src). You can test the result of the comparison by checking the appropriate flags in the flags register. For details on how this is done, see "The "Set on Condition" Instructions" on page 281 and "The Conditional Jump Instructions" on page 296.

Usually you'll want to execute a conditional jump instruction after a cmp instruction. This two step process, comparing two values and setting the flag bits then testing the flag bits with the conditional jump instructions, is a very efficient mechanism for making decisions in a program.

Probably the first place to start when exploring the cmp instruction is to take a look at exactly how the cmp instruction affects the flags. Consider the following cmp instruction:

```
                cmp     ax, bx
```

This instruction performs the computation ax-bx and sets the flags depending upon the result of the computation. The flags are set as follows:

Z: The zero flag is set if and only if ax = bx. This is the only time ax-bx produces a zero result. Hence, you can use the zero flag to test for equality or inequality.

S: The sign flag is set to one if the result is negative. At first glance, you might think that this flag would be set if ax is less than bx but this isn't always the case. If ax=7FFFh and bx=-1 (0FFFFh) subtracting ax from bx produces 8000h, which is negative (and so the sign flag will be set). So, for signed comparisons anyway, the sign flag doesn't contain the proper status. For unsigned operands, consider ax=0FFFFh and bx=1. Ax is greater than bx but their difference is 0FFFEh which is still negative. As it turns out, the sign flag and the overflow flag, taken together, can be used for comparing two signed values.

O: The overflow flag is set after a cmp operation if the difference of ax and bx produced an overflow or underflow. As mentioned above, the sign flag and the overflow flag are both used when performing signed comparisons.

C: The carry flag is set after a cmp operation if subtracting bx from ax requires a borrow. This occurs only when ax is less than bx where ax and bx are both unsigned values.

The cmp instruction also affects the parity and auxiliary carry flags, but you'll rarely test these two flags after a compare operation. Given that the cmp instruction sets the flags in this fashion, you can test the comparison of the two operands with the following flags:

```
cmp Oprnd1, Oprnd2
```

Table 27: Condition Code Settings After CMP

Unsigned operands:	Signed operands:
Z: equality/inequality	Z: equality/inequality
C: Oprnd1 < Oprnd2 (C=1) Oprnd1 >= Oprnd2 (C=0)	C: no meaning
S: no meaning	S: see below
O: no meaning	O: see below

For signed comparisons, the S (sign) and O (overflow) flags, taken together, have the following meaning:
If ((S=0) and (O=1)) or ((S=1) and (O=0)) then Oprnd1 < Oprnd2 when using a signed comparison.
If ((S=0) and (O=0)) or ((S=1) and (O=1)) then Oprnd1 >= Oprnd2 when using a signed comparison.

To understand why these flags are set in this manner, consider the following examples:

```
Oprnd1              minus       Oprnd2          S     O
------                          ------          -     -

0FFFF (-1)            -         0FFFE (-2)      0     0
08000                 -         00001           0     1
0FFFE (-2)            -         0FFFF (-1)      1     0
07FFF (32767)         -         0FFFF (-1)      1     1
```

Remember, the cmp operation is really a subtraction, therefore, the first example above computes (-1)-(-2) which is (+1). The result is positive and an overflow did not occur so both the S and O flags are zero. Since (S xor O) is zero, Oprnd1 is greater than or equal to Oprnd2.

In the second example, the cmp instruction would compute (-32768)-(+1) which is (-32769). Since a 16-bit signed integer cannot represent this value, the value wraps around to 7FFFh (+32767) and sets the overflow flag. Since the result is positive (at least within the confines of 16 bits) the sign flag is cleared. Since (S xor O) is one here, Oprnd1 is less than Oprnd2.

In the third example above, cmp computes (-2)-(-1) which produces (-1). No overflow occurred so the O flag is zero, the result is negative so the sign flag is one. Since (S xor O) is one, Oprnd1 is less than Oprnd2.

In the fourth (and final) example, cmp computes (+32767)-(-1). This produces (+32768), setting the overflow flag. Furthermore, the value wraps around to 8000h (-32768) so the sign flag is set as well. Since (S xor O) is zero, Oprnd1 is greater than or equal to Oprnd2.

6.5.4 The CMPXCHG, and CMPXCHG8B Instructions

The cmpxchg (compare and exchange) instruction is available only on the 80486 and later processors. It supports the following syntax:

```
cmpxchg     reg, reg
cmpxchg     mem, reg
```

The operands must be the same size (eight, sixteen, or thirty-two bits). This instruction also uses the accumulator register; it automatically chooses al, ax, or eax to match the size of the operands.

This instruction compares al, ax, or eax with the first operand and sets the zero flag if they are equal. If so, then cmpxchg copies the second operand into the first. If they are not equal, cmpxchg copies the first operand into the accumulator. The following algorithm describes this operation:

```
cmpxchg             operand1, operand2
if ({al/ax/eax} = operand1) then[8]
          zero := 1                         ;Set the zero flag
          operand1 := operand2
else
          zero := 0                         ;Clear the zero flag
          {al/ax/eax} := operand1
endif
```

Cmpxchg supports certain operating system data structures requiring atomic operations[9] and semaphores. Of course, if you can fit the above algorithm into your code, you can use the cmpxchg instruction as appropriate.

Note: unlike the cmp instruction, the cmpxchg instruction only affects the 80x86 zero flag. You cannot test other flags after cmpxchg as you could with the cmp instruction.

The Pentium processor supports a 64 bit compare and exchange instruction – cmpxchg8b. It uses the syntax:

```
cmpxchg8b   ax, mem64
```

This instruction compares the 64 bit value in edx:eax with the memory value. If they are equal, the Pentium stores ecx:ebx into the memory location, otherwise it loads edx:eax with the memory location. This instruction sets the zero flag according to the result. It does not affect any other flags.

6.5.5 The NEG Instruction

The neg (negate) instruction takes the two's complement of a byte or word. It takes a single (destination) operation and negates it. The syntax for this instruction is

```
neg       dest
```

It computes the following:

```
dest := 0 - dest
```

This effectively reverses the sign of the destination operand.

8. The choice of al, ax, or eax is made by the size of the operands. Both operands to cmpxchg must be the same size.

9. An atomic operation is one that the system cannot interrupt.

If the operand is zero, its sign does not change, although this clears the carry flag. Negating any other value sets the carry flag. Negating a byte containing -128, a word containing -32,768, or a double word containing -2,147,483,648 does not change the operand, but will set the overflow flag. Neg always updates the A, S, P, and Z flags as though you were using the sub instruction.

The allowable forms are:

```
neg     reg
neg     mem
```

The operands may be eight, sixteen, or (on the 80386 and later) thirty-two bit values.

Some examples:

```
; J := - J
                neg     J

; J := -K
                mov     ax, K
                neg     ax
                mov     J, ax
```

6.5.6 The Multiplication Instructions: MUL, IMUL, and AAM

The multiplication instructions provide you with your first taste of irregularity in the 8086's instruction set. Instructions like add, adc, sub, sbb, and many others in the 8086 instruction set use a mod-reg-r/m byte to support two operands. Unfortunately, there aren't enough bits in the 8086's opcode byte to support all instructions, so the 8086 uses the reg bits in the mod-reg-r/m byte as an opcode extension. For example, inc, dec, and neg do not require two operands, so the 80x86 CPUs use the reg bits as an extension to the eight bit opcode. This works great for single operand instructions, allowing Intel's designers to encode several instructions (eight, in fact) with a single opcode.

Unfortunately, the multiply instructions require special treatment and Intel's designers were still short on opcodes, so they designed the multiply instructions to use a single operand. The reg field contains an opcode extension rather than a register value. Of course, multiplication *is* a two operand function. The 8086 always assumes the accumulator (al, ax, or eax) is the destination operand. This irregularity makes using multiplication on the 8086 a little more difficult than other instructions because one operand has to be in the accumulator. Intel adopted this unorthogonal approach because they felt that programmers would use multiplication far less often than instructions like add and sub.

One problem with providing only a mod-reg-r/m form of the instruction is that you cannot multiply the accumulator by a constant; the mod-reg-r/m byte does not support the immediate addressing mode. Intel quickly discovered the need to support multiplication by a constant and provide some support for this in the 80286 processor[10]. This was especially important for multidimensional array access. By the time the 80386 rolled around, Intel generalized one form of the multiplication operation allowing standard mod-reg-r/m operands.

There are two forms of the multiply instruction: an unsigned multiplication (mul) and a signed multiplication (imul). Unlike addition and subtraction, you need separate instructions for these two operations.

The multiply instructions take the following forms:

10. On the original 8086 chip multiplication by a constant was always faster using shifts, additions, and subtractions. Perhaps Intel's designers didn't bother with multiplication by a constant for this reason. However, the 80286 multiply instruction was faster than the 8086 multiply instruction, so it was no longer true that multiplication was slower and the corresponding shift, add, and subtract instructions.

Unsigned Multiplication:

```
mul     reg
mul     mem
```

Signed (Integer) Multiplication:

```
imul    reg
imul    mem
imul    reg, reg, immediate          (2)
imul    reg, mem, immediate          (2)
imul    reg, immediate               (2)
imul    reg, reg                     (3)
imul    reg, mem                     (3)
```

BCD Multiplication Operations:

```
aam
```

```
2- Available on the 80286 and later, only.
3- Available on the 80386 and later, only.
```

As you can see, the multiply instructions are a real mess. Worse yet, you have to use an 80386 or later processor to get near full functionality. Finally, there are some restrictions on these instructions not obvious above. Alas, the only way to deal with these instructions is to memorize their operation.

Mul, available on all processors, multiplies unsigned eight, sixteen, or thirty-two bit operands. Note that when multiplying two n-bit values, the result may require as many as 2*n bits. Therefore, if the operand is an eight bit quantity, the result will require sixteen bits. Likewise, a 16 bit operand produces a 32 bit result and a 32 bit operand requires 64 bits for the result.

The mul instruction, with an eight bit operand, multiplies the al register by the operand and stores the 16 bit result in ax. So

```
mul     operand8
```
or
```
imul    operand8
```

computes:

```
ax := al * operand8
```

"*" represents an unsigned multiplication for mul and a signed multiplication for imul.

If you specify a 16 bit operand, then mul and imul compute:

```
dx:ax := ax * operand16
```

"*" has the same meanings as above and dx:ax means that dx contains the H.O. word of the 32 bit result and ax contains the L.O. word of the 32 bit result.

If you specify a 32 bit operand, then mul and imul compute the following:

```
edx:eax := eax * operand32
```

"*" has the same meanings as above and edx:eax means that edx contains the H.O. double word of the 64 bit result and eax contains the L.O. double word of the 64 bit result.

If an 8x8, 16x16, or 32x32 bit product requires more than eight, sixteen, or thirty-two bits (respectively), the mul and imul instructions set the carry and overflow flags.

Mul and imul scramble the A, P, S, and Z flags. Especially note that the sign and zero flags do not contain meaningful values after the execution of these two instructions.

Imul (integer multiplication) operates on signed operands. There are many different forms of this instruction as Intel attempted to generalize this instruction with successive processors. The previous paragraphs describe the first form of the imul instruction, with a single operand. The next three forms of the imul instruction are available only on the 80286 and later processors. They provide the ability to multiply a register by an immediate value. The last two forms, available only on 80386 and later processors, provide the ability to multiply an arbitrary register by another register or memory location. Expanded to show allowable operand sizes, they are

```
imul      operand1, operand2, immediate     ;General form

imul      reg16, reg16, immediate8
imul      reg16, reg16, immediate16
imul      reg16, mem16, immediate8
imul      reg16, mem16, immediate16
imul      reg16, immediate8
imul      reg16, immediate16
imul      reg32, reg32, immediate8          (3)
imul      reg32, reg32, immediate32         (3)
imul      reg32, mem32, immediate8          (3)
imul      reg32, mem32, immediate32         (3)
imul      reg32, immediate8                 (3)
imul      reg32, immediate32                (3)
```

3- Available on the 80386 and later, only.

The imul reg, immediate instructions are a special syntax the assembler provides. The encodings for these instructions are the same as imul reg, reg, immediate. The assembler simply supplies the same register value for both operands.

These instructions compute:

```
operand1 := operand2 * immediate
operand1 := operand1 * immediate
```

Besides the number of operands, there are several differences between these forms and the single operand mul/imul instructions:

- There isn't an 8x8 bit multiplication available (the $immediate_8$ operands simply provide a shorter form of the instruction. Internally, the CPU sign extends the operand to 16 or 32 bits as necessary).
- These instructions do not produce a 2*n bit result. That is, a 16x16 multiply produces a 16 bit result. Likewise, a 32x32 bit multiply produces a 32 bit result. These instructions set the carry and overflow flags if the result does not fit into the destination register.
- The 80286 version of imul allows an immediate operand, the standard mul/imul instructions do not.

The last two forms of the imul instruction are available only on 80386 and later processors. With the addition of these formats, the imul instruction is *almost* as general as the add instruction[11]:

```
imul      reg, reg
imul      reg, mem
```

These instructions compute

```
reg := reg * reg
reg := reg * mem
```

and

Both operands must be the same size. Therefore, like the 80286 form of the imul instruction, you must test the carry or overflow flag to detect overflow. If overflow does occur, the CPU loses the H.O. bits of the result.

Important Note: Keep in mind that the zero flag contains an indeterminate result after executing a multiply instruction. You cannot test the zero flag to see if the result is zero after a multiplication. Likewise, these instructions scramble the sign flag. If you need to check these flags, compare the result to zero after testing the carry or overflow flags.

The aam (ASCII Adjust after Multiplication) instruction, like aaa and aas, lets you adjust an unpacked decimal value after multiplication. This instruction operates directly on the ax register. It assumes that you've multiplied two eight bit values in the range 0..9 together and the result is sitting in ax (actually, the result will be sitting in al since 9*9 is 81, the largest possible value; ah must contain zero). This instruction divides ax by 10 and leaves the quotient in ah and the remainder in al:

11. There are still some restrictions on the size of the operands, e.g., no eight bit registers, you have to consider.

```
ah := ax div 10
al := ax mod 10
```

Unlike the other decimal/ASCII adjust instructions, assembly language programs regularly use aam since conversion between number bases uses this algorithm.

Note: the aam instruction consists of a two byte opcode, the second byte of which is the immediate constant 10. Assembly language programmers have discovered that if you substitute another immediate value for this constant, you can change the divisor in the above algorithm. This, however, is an undocumented feature. It works in all varieties of the processor Intel has produced to date, but there is no guarantee that Intel will support this in future processors. Of course, the 80286 and later processors let you multiply by a constant, so this trick is hardly necessary on modern systems.

There is no dam (decimal adjust for multiplication) instruction on the 80x86 processor.

Perhaps the most common use of the imul instruction is to compute offsets into multi-dimensional arrays. Indeed, this is probably the main reason Intel added the ability to multiply a register by a constant on the 80286 processor. In Chapter Four, this text used the standard 8086 mul instruction for array index computations. However, the extended syntax of the imul instruction makes it a much better choice as the following examples demonstrate:

```
MyArray         word    8 dup ( 7 dup ( 6 dup (?)))    ;8x7x6 array.
J               word    ?
K               word    ?
M               word    ?
                 .
                 .
                 .
; MyArray [J, K, M] := J + K - M

                mov     ax, J
                add     ax, K
                sub     ax, M

                mov     bx, J       ;Array index :=
                imul    bx, 7       ;          ((J*7 + K) * 6 + M) * 2
                add     bx, K
                imul    bx, 6
                add     bx, M
                add     bx, bx      ;BX := BX * 2

                mov     MyArray[bx], ax
```

Don't forget that the multiplication instructions are very slow; often an order of magnitude slower than an addition instruction. There are faster ways to multiply a value by a constant. See "Multiplying Without MUL and IMUL" on page 487 for all the details.

6.5.7 The Division Instructions: DIV, IDIV, and AAD

The 80x86 divide instructions perform a 64/32 division (80386 and later only), a 32/16 division or a 16/8 division. These instructions take the form:

```
div     reg         For unsigned division
div     mem

idiv    reg         For signed division
idiv    mem

aad                 ASCII adjust for division
```

The div instruction computes an unsigned division. If the operand is an eight bit operand, div divides the ax register by the operand leaving the quotient in al and the remainder (modulo) in ah. If the operand is a 16 bit quantity, then the div instruction divides the 32 bit quantity in dx:ax by the operand leaving the quotient in ax and the remainder in . With 32 bit operands (on the 80386 and later) div divides the 64 bit value in edx:eax by the operand leaving the quotient in eax and the remainder in edx.

You cannot, on the 80x86, simply divide one eight bit value by another. If the denominator is an eight bit value, the numerator must be a sixteen bit value. If you need to divide one unsigned eight bit value by another, you must zero extend the numerator to sixteen bits. You can accomplish this by loading the numerator into the al register and then moving zero into the ah register. Then you can divide ax by the denominator operand to produce the correct result. *Failing to zero extend al before executing div may cause the 80x86 to produce incorrect results!*

When you need to divide two 16 bit unsigned values, you must zero extend the ax register (which contains the numerator) into the dx register. Just load the immediate value zero into the dx register[12]. If you need to divide one 32 bit value by another, you must zero extend the eax register into edx (by loading a zero into edx) before the division.

When dealing with signed integer values, you will need to sign extend al to ax, ax to dx or eax into edx before executing idiv. To do so, use the cbw, cwd, cdq, or movsx instructions. If the H.O. byte or word does not already contain significant bits, then you must sign extend the value in the accumulator (al/ax/eax) before doing the idiv operation. Failure to do so may produce incorrect results.

There is one other catch to the 80x86's divide instructions: you can get a fatal error when using this instruction. First, of course, you can attempt to divide a value by zero. Furthermore, the quotient may be too large to fit into the eax, ax, or al register. For example, the 16/8 division "8000h / 2" produces the quotient 4000h with a remainder of zero. 4000h will not fit into eight bits. If this happens, or you attempt to divide by zero, the 80x86 will generate an *int 0* trap. This usually means BIOS will print "division by zero" or "divide error" and abort your program. If this happens to you, chances are you didn't sign or zero extend your numerator before executing the division operation. Since this error will cause your program to crash, you should be very careful about the values you select when using division.

The auxiliary carry, carry, overflow, parity, sign, and zero flags are undefined after a division operation. If an overflow occurs (or you attempt a division by zero) then the 80x86 executes an INT 0 (interrupt zero).

Note that the 80286 and later processors do not provide special forms for idiv as they do for imul. Most programs use division far less often than they use multiplication, so Intel's designers did not bother creating special instructions for the divide operation. Note that there is no way to divide by an immediate value. You must load the immediate value into a register or a memory location and do the division through that register or memory location.

The aad (ASCII Adjust before Division) instruction is another unpacked decimal operation. It splits apart unpacked binary coded decimal values before an ASCII division operation. Although this text will not cover BCD arithmetic, the aad instruction is useful for other operations. The algorithm that describes this instruction is

```
al := ah*10 + al
ah := 0
```

This instruction is quite useful for converting strings of digits into integer values (see the questions at the end of this chapter).

The following examples show how to divide one sixteen bit value by another.

```
; J := K / M (unsigned)

                mov     ax, K       ;Get dividend
                mov     dx, 0       ;Zero extend unsigned value in AX to DX.

        < In practice, we should verify that M does not contain zero here >

                div     M
                mov     J, ax

; J := K / M (signed)
```

12. Or use the MOVZX instruction on the 80386 and later processors.

```
                mov     ax, K       ;Get dividend
                cwd                 ;Sign extend signed value in AX to DX.
        < In practice, we should verify that M does not contain zero here >
                idiv    M
                mov     J, ax
; J := (K*M)/P
                mov     ax, K       ;Note that the imul instruction produces
                imul    M           ; a 32 bit result in DX:AX, so we don't
                idiv    P           ; need to sign extend AX here.
                mov     J, ax       ;Hope and pray result fits in 16 bits!
```

6.6 Logical, Shift, Rotate and Bit Instructions

The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror, rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar. The 80386 and later processors provide an even richer set of operations. These are bt, bts, btr, btc, bsf, bsr, shld, shrd, and the conditional set instructions (set*cc*).

These instructions can manipulate bits, convert values, do logical operations, pack and unpack data, and do arithmetic operations. The following sections describe each of these instructions in detail.

6.6.1 The Logical Instructions: AND, OR, XOR, and NOT

The 80x86 logical instructions operate on a bit-by-bit basis. Both eight, sixteen, and thirty-two bit versions of each instruction exist. The and, not, or, and xor instructions do the following:

```
        and     dest, source                ;dest := dest and source
        or      dest, source                ;dest := dest or source
        xor     dest, source                ;dest := dest xor source
        not     dest                        ;dest := not dest
```

The specific variations are

```
        and     reg, reg
        and     mem, reg
        and     reg, mem
        and     reg, immediate data
        and     mem, immediate data
        and     eax/ax/al, immediate data

        or uses the same formats as AND
        xor uses the same formats as AND

        not     register
        not     mem
```

Except not, these instructions affect the flags as follows:

- They clear the carry flag.
- They clear the overflow flag.
- They set the zero flag if the result is zero, they clear it otherwise.
- They copy the H.O. bit of the result into the sign flag.
- They set the parity flag according to the parity (number of one bits) in the result.
- They scramble the auxiliary carry flag.

The not instruction does not affect any flags.

Testing the zero flag after these instructions is particularly useful. The and instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions (since this would produce a zero result); for example, if the source operand contained a

single one bit, then the zero flag will be set if the corresponding destination bit is zero, it will be one otherwise. The or instruction will only set the zero flag if both operands contain zero. The xor instruction will set the zero flag only if both operands are equal. Notice that the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form

```
xor     reg16, reg16
```

is shorter than the comparable `mov reg,0` instruction.

Like the addition and subtraction instructions, the and, or, and xor instructions provide special forms involving the accumulator register and immediate data. These forms are shorter and sometimes faster than the general "register, immediate" forms. Although one does not normally think of operating on signed data with these instructions, the 80x86 does provide a special form of the "reg/mem, immediate" instructions that sign extend a value in the range -128..+127 to sixteen or thirty-two bits, as necessary.

The instruction's operands must all be the same size. On pre-80386 processors they can be eight or sixteen bits. On 80386 and later processors, they may be 32 bits long as well. These instructions compute the obvious bitwise logical operation on their operands, see Chapter One for details on these operations.

You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; see for more details. Likewise, you can use the or instruction to force certain bits to one in the destination operand; see "Masking Operations with the OR Instruction" on page 491 for the details. You can use these instructions, along with the shift and rotate instructions described next, to pack and unpack data. See "Packing and Unpacking Data Types" on page 491 for more details.

6.6.2 The Shift Instructions: SHL/SAL, SHR, SAR, SHLD, and SHRD

The 80x86 supports three different shift instructions (shl and sal are the same instruction): shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right). The 80386 and later processors provide two additional shifts: shld and shrd.

The shift instructions move bits around in a register or memory location. The general format for a shift instruction is

```
shl     dest, count
sal     dest, count
shr     dest, count
sar     dest, count
```

Dest is the value to shift and count specifies the number of bit positions to shift. For example, the shl instruction shifts the bits in the destination operand to the left the number of bit positions specified by the count operand. The shld and shrd instructions use the format:

```
shld    dest, source, count
shrd    dest, source, count
```

The specific forms for these instructions are

```
shl     reg, 1
shl     mem, 1
shl     reg, imm            (2)
shl     mem, imm            (2)
shl     reg, cl
shl     mem, cl

sal is a synonym for shl and uses the same formats.
shr uses the same formats as shl.
sar uses the same formats as shl.
```

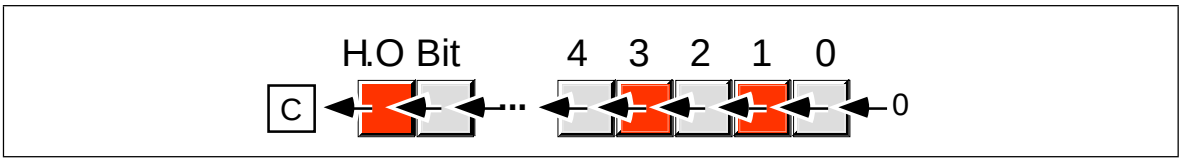


Figure 6.2 Shift Left Operation

```
shld      reg, reg, imm         (3)
shld      mem, reg, imm         (3)
shld      reg, reg, cl          (3)
shld      mem, reg, cl          (3)
```

shrd uses the same formats as shld.

```
2- This form is available on 80286 and later processors only.
3- This form is available on 80386 and later processors only.
```

For 8088 and 8086 CPUs, the number of bits to shift is either “1” or the value in cl. On 80286 and later processors you can use an eight bit immediate constant. Of course, the value in cl or the immediate constant should be less than or equal to the number of bits in the destination operand. It would be a waste of time to shift left al by nine bits (eight would produce the same result, as you will soon see). Algorithmically, you can think of the shift operations with a count other than one as follows:

```
for temp := 1 to count do
          shift dest, 1
```

There are minor differences in the way the shift instructions treat the overflow flag when the count is not one, but you can ignore this most of the time.

The shl, sal, shr, and sar instructions work on eight, sixteen, and thirty-two bit operands. The shld and shrd instructions work on 16 and 32 bit destination operands only.

6.6.2.1 SHL/SAL

The shl and sal mnemonics are synonyms. They represent the same instruction and use identical binary encodings. These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag (see Figure 6.2).

The shl/sal instruction sets the condition code bits as follows:

- If the shift count is zero, the shl instruction doesn’t affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- The overflow flag will contain one if the two H.O. bits were different prior to a single bit shift. The overflow flag is undefined if the shift count is not one.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

```
shl       ah, 4         ;This form requires an 80286 or later
or        al, ah        ;Merge in H.O. four bits.
```

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction). If the

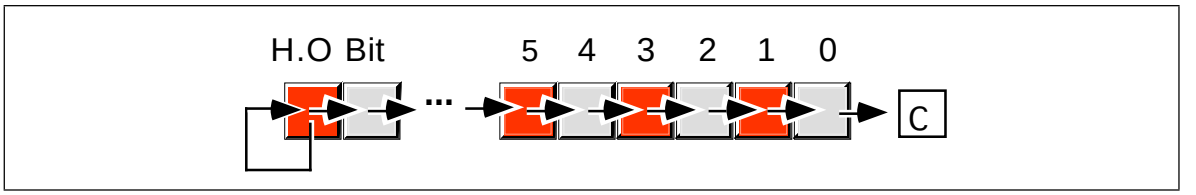


Figure 6.3 Arithmetic Shift Right Operation

H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

```
shl     ah, 4       ;Move L.O. bits to H.O. position.
and     al, 0Fh     ;Clear H.O. four bits.
or      al, ah      ;Merge the bits.
```

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the shift left instruction for multiplication by powers of two:

```
shl     ax, 1       ;Equivalent to AX*2
shl     ax, 2       ;Equivalent to AX*4
shl     ax, 3       ;Equivalent to AX*8
shl     ax, 4       ;Equivalent to AX*16
shl     ax, 5       ;Equivlaent to AX*32
shl     ax, 6       ;Equivalent to AX*64
shl     ax, 7       ;Equivalent to AX*128
shl     ax, 8       ;Equivalent to AX*256
etc.
```

Note that shl ax, 8 is equivalent to the following two instructions:

```
mov     ah, al
mov     al, 0
```

The shl/sal instruction multiplies both signed and unsigned values by two for each shift. This instruction sets the carry flag if the result does not fit in the destination operand (i.e., unsigned overflow occurs). Likewise, this instruction sets the overflow flag if the signed result does not fit in the destination operation. This occurs when you shift a zero into the H.O. bit of a negative number or you shift a one into the H.O. bit of a non-negative number.

6.6.2.2 SAR

The sar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit (see Figure 6.3).

The sar instruction sets the flag bits as follows:

- If the shift count is zero, the sar instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- The overflow flag will contain zero if the shift count is one. Overflow can never occur with this instruction. However, if the count is not one, the value of the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The auxiliary carry flag is always undefined after the sar instruction.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

```
sar     ax, 1       ;Signed division by 2
sar     ax, 2       ;Signed division by 4
sar     ax, 3       ;Signed division by 8
sar     ax, 4       ;Signed division by 16
sar     ax, 5       ;Signed division by 32
sar     ax, 6       ;Signed division by 64
sar     ax, 7       ;Signed division by 128
sar     ax, 8       ;Signed division by 256
```

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is negative, idiv truncates towards zero while sar truncates towards negative infinity. The following examples demonstrate the difference:

```
mov     ax, -15
cwd
mov     bx, 2
idiv                ;Produces -7

mov     ax, -15
sar     ax, 1       ;Produces -8
```

Keep this in mind if you use sar for integer division operations.

The sar ax, 8 instruction effectively copies ah into al and then sign extends al into ax. This is because sar ax, 8 will shift ah down into al but leave a copy of ah's H.O. bit in all the bit positions of ah. Indeed, you can use the sar instruction on 80286 and later processors to sign extend one register into another. The following code sequences provide examples of this usage:

```
; Equivalent to CBW:

                mov     ah, al
                sar     ah, 7

; Equivalent to CWD:

                mov     dx, ax
                sar     dx, 15

; Equivalent to CDQ:

                mov     edx, eax
                sar     edx, 31
```

Of course it may seem silly to use two instructions where a single instruction might suffice; however, the cbw, cwd, and cdq instructions only sign extend al into ax, ax into dx:ax, and eax into edx:eax. Likewise, the movsx instruction copies its sign extended operand into a destination operand twice the size of the source operand. The sar instruction lets you sign extend one register into another register of the same size, with the second register containing the sign extension bits:

```
; Sign extend bx into cx:bx

                mov     cx, bx
                sar     cx, 15
```

6.6.2.3 SHR

The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit (see Figure 6.4).

The shr instruction sets the flag bits as follows:

- If the shift count is zero, the shr instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the operand.
- If the shift count is one, the overflow flag will contain the value of the H.O. bit of the operand prior to the shift (i.e., this instruction sets the

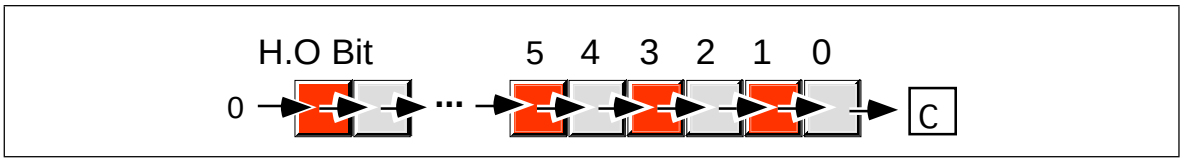


Figure 6.4 Shift Right Operation

overflow flag if the sign changes). However, if the count is not one, the value of the overflow flag is undefined.

- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result, which is always zero.
- The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.
- The auxiliary carry flag is always undefined after the shr instruction.

The shift right instruction is especially useful for unpacking data. For example, suppose you want to extract the two nibbles in the al register, leaving the H.O. nibble in ah and the L.O. nibble in al. You could use the following code to do this:

```
mov     ah, al      ;Get a copy of the H.O. nibble
shr     ah, 4       ;Move H.O. to L.O. and clear H.O. nibble
and     al, 0Fh     ;Remove H.O. nibble from al
```

Since shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

```
shr     ax, 1       ;Equivalent to AX/2
shr     ax, 2       ;Equivalent to AX/4
shr     ax, 3       ;Equivalent to AX/8
shr     ax, 4       ;Equivalent to AX/16
shr     ax, 5       ;Equivlaent to AX/32
shr     ax, 6       ;Equivalent to AX/64
shr     ax, 7       ;Equivalent to AX/128
shr     ax, 8       ;Equivalent to AX/256
etc.
```

Note that shr ax, 8 is equivalent to the following two instructions:

```
mov     al, ah
mov     ah, 0
```

Remember that division by two using shr only works for *unsigned* operands. If ax contains -1 and you execute shr ax, 1 the result in ax will be 32767 (7FFFh), not -1 or zero as you would expect. Use the sar instruction if you need to divide a signed integer by some power of two.

6.6.2.4 The SHLD and SHRD Instructions

The shld and shrd instructions provide double precision shift left and right operations, respectively. These instructions are available only on 80386 and later processors. Their generic forms are

```
shld    operand1, operand2, immediate
shld    operand1, operand2, cl
shrd    operand1, operand2, immediate
shrd    operand1, operand2, cl
```

$Operand_2$ must be a sixteen or thirty-two bit register. $Operand_1$ can be a register or a memory location. Both operands must be the same size. The immediate operand can be a value in the range zero through n-1, where n is the number of bits in the two operands; it specifies the number of bits to shift.

The shld instruction shifts bits in $operand_1$ to the left. The H.O. bit shifts into the carry flag and the H.O. bit of $operand_2$ shifts into the L.O. bit of $operand_1$. Note that this instruc-

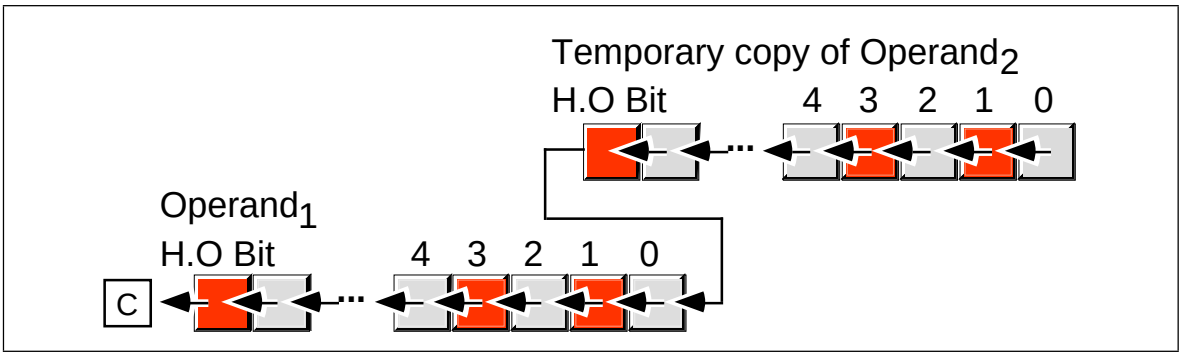


Figure 6.5 Double Precision Shift Left Operation

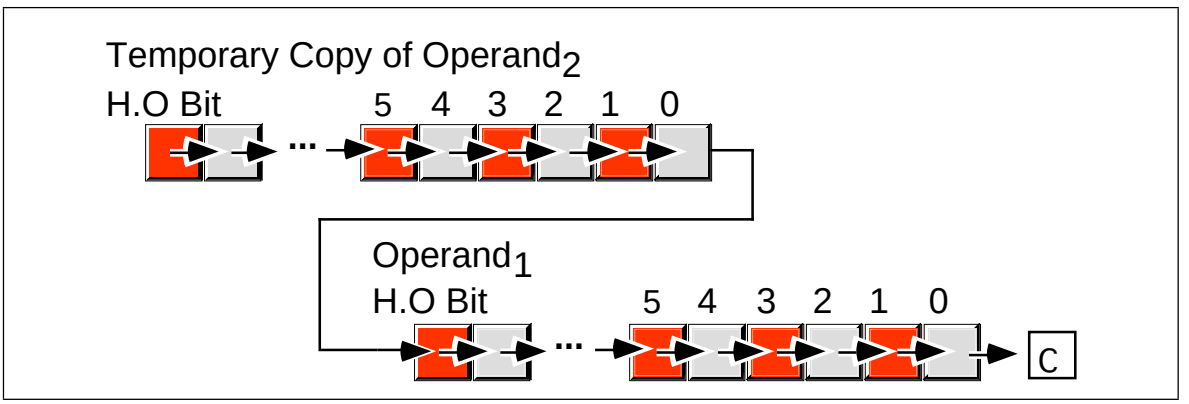


Figure 6.6 Double Precision Shift Right Operation

tion does not modify the value of operand$_2$, it uses a temporary copy of operand$_2$ during the shift. The immediate operand specifies the number of bits to shift. If the count is *n*, then shld shifts bit *n*-1 into the carry flag. It also shifts the H.O. *n* bits of operand$_2$ into the L.O. *n* bits of operand$_1$. Pictorially, the shld instruction appears in Figure 6.5.

The shld instruction sets the flag bits as follows:

- If the shift count is zero, the shld instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the operand$_1$.
- If the shift count is one, the overflow flag will contain one if the sign bit of operand$_1$ changes during the shift. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

The shld instruction is useful for packing data from many different sources. For example, suppose you want to create a word by merging the H.O. nibbles of four other words. You could do this with the following code:

```
mov     ax, Value4   ;Get H.O. nibble
shld    bx, ax, 4    ;Copy H.O. bits of AX to BX.
mov     ax, Value3   ;Get nibble #2.
shld    bx, ax, 4    ;Merge into bx.
mov     ax, Value2   ;Get nibble #1.
shld    bx, ax, 4    ;Merge into bx.
mov     ax, Value1   ;Get L.O. nibble
shld    bx, ax, 4    ;BX now contains all four nibbles.
```

The shrd instruction is similar to shld except, of course, it shifts its bits right rather than left. To get a clear picture of the shrd instruction, consider Figure 6.6.

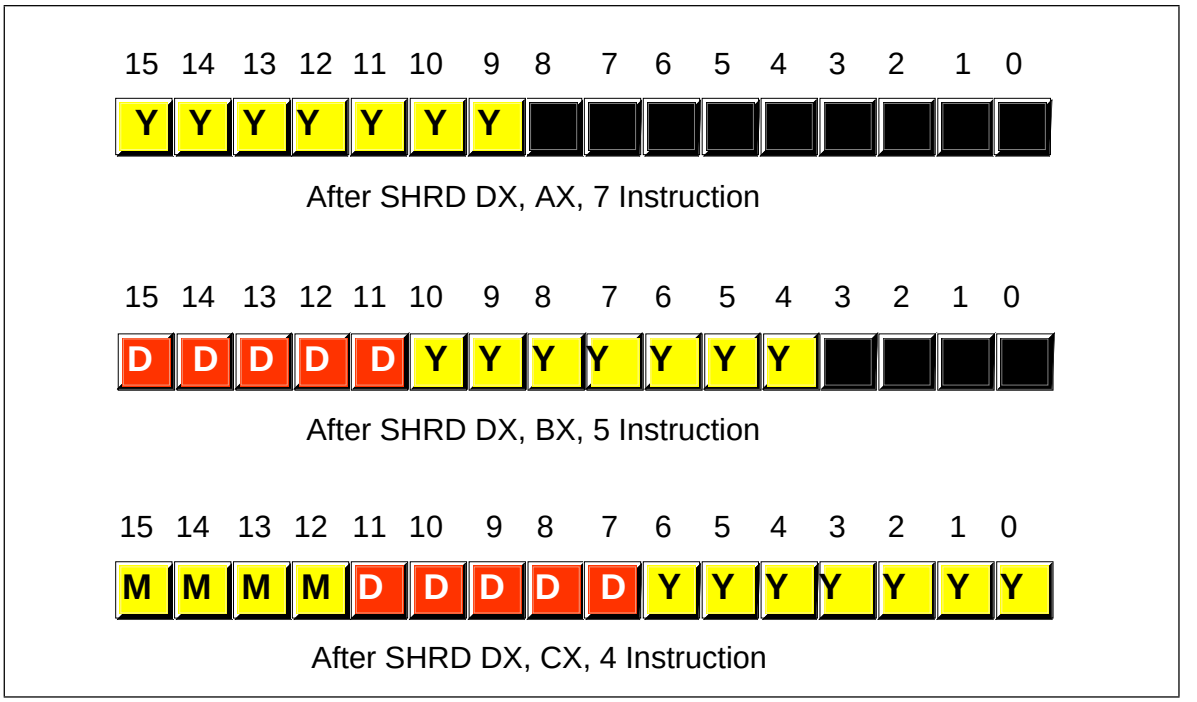


Figure 6.7 Packing Data with an SHRD Instruction

The shrd instruction sets the flag bits as follows:

- If the shift count is zero, the shrd instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the operand$_1$.
- If the shift count is one, the overflow flag will contain one if the H.O. bit of operand$_1$ changes. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

Quite frankly, these two instructions would probably be slightly more useful if Operand$_2$ could be a memory location. Intel designed these instructions to allow fast multiprecision (64 bits, or more) shifts. For more information on such usage, see "Extended Precision Shift Operations" on page 482.

The shrd instruction is marginally more useful than shld for packing data. For example, suppose that ax contains a value in the range 0..99 representing a year (1900..1999), bx contains a value in the range 1..31 representing a day, and cx contains a value in the range 1..12 representing a month (see "Bit Fields and Packed Data" on page 28). You can easily use the shrd instruction to pack this data into dx as follows:

```
shrd    dx, ax, 7
shrd    dx, bx, 5
shrd    dx, cx, 4
```

See Figure 6.7 for a blow-by-blow example.

6.6.3 The Rotate Instructions: RCL, RCR, ROL, and ROR

The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions recirculate through the operand. They include rcl (rotate through carry left), rcr (rotate through carry right), rol (rotate left), and ror (rotate right). These instructions all take the forms:

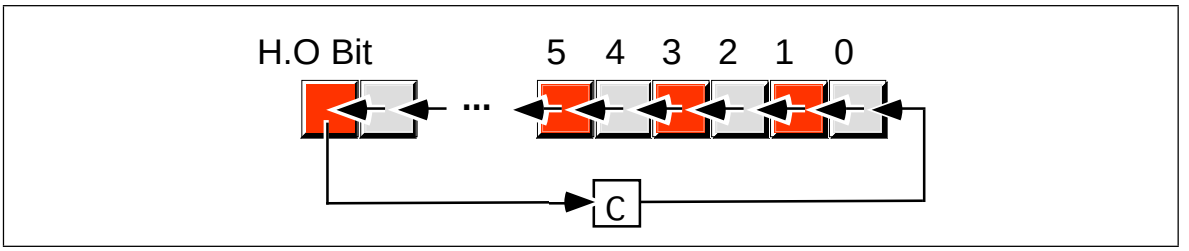


Figure 6.8 Rotate Through Carry Left Operation

```
rcl       dest, count
rol       dest, count
rcr       dest, count
ror       dest, count
```

The specific forms are

```
rcl       reg, 1
rcl       mem, 1
rcl       reg, imm    (2)
rcl       mem, imm    (2)
rcl       reg, cl
rcl       mem, cl

rol uses the same formats as rcl.
rcr uses the same formats as rcl.
ror uses the same formats as rcl.

2- This form is avialable on 80286 and later processors only.
```

6.6.3.1 RCL

The rcl (rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right (see Figure 6.8).

Note that if you rotate through carry an object *n+1* times, where *n* is the number of bits in the object, you wind up with your original value. Keep in mind, however, that some flags may contain different values after *n+1* rcl operations.

The rcl instruction sets the flag bits as follows:

- The carry flag contains the last bit shifted out of the H.O. bit of the operand.
- If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.
- The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.

Important warning: unlike the shift instructions, the rotate instructions do not affect the sign, zero, parity, or auxiliary carry flags. This lack of orthogonality can cause you lots of grief if you forget it and attempt to test these flags after an rcl operation. If you need to test one of these flags after an rcl operation, test the carry and overflow flags first (if necessary) then compare the result to zero to set the other flags.

6.6.3.2 RCR

The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit (see Figure 6.9).

This instruction sets the flags in a manner analogous to rcl:

- The carry flag contains the last bit shifted out of the L.O. bit of the operand.

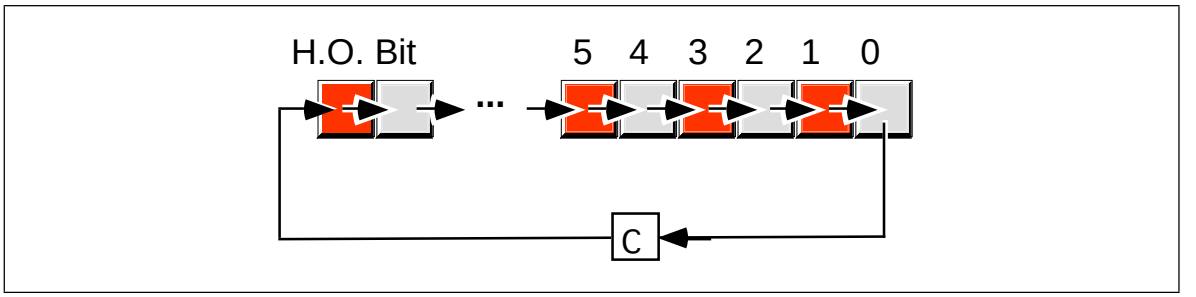


Figure 6.9 Rotate Through Carry Right Operation

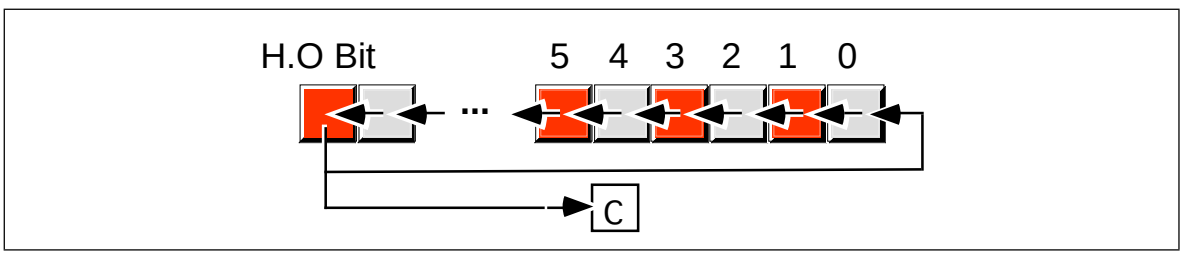


Figure 6.10 Rotate Left Operation

- If the shift count is one, then rcr sets the overflow flag if the sign changes (meaning the values of the H.O. bit and carry flag were not the same before the execution of the instruction). However, if the count is not one, the value of the overflow flag is undefined.
- The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.

Keep in mind the warning given for rcl above.

6.6.3.3 ROL

The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit, rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag (see Figure 6.10).

The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction. **Don't forget the warning about the flags!**

Like shl, the rol instruction is often useful for packing and unpacking data. For example, suppose you want to extract bits 10..14 in ax and leave these bits in bits 0..4. The following code sequences will both accomplish this:

```
shr     ax, 10
and     ax, 1Fh

rol     ax, 6
and     ax, 1Fh
```

6.6.3.4 ROR

The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit (see Figure 6.11).

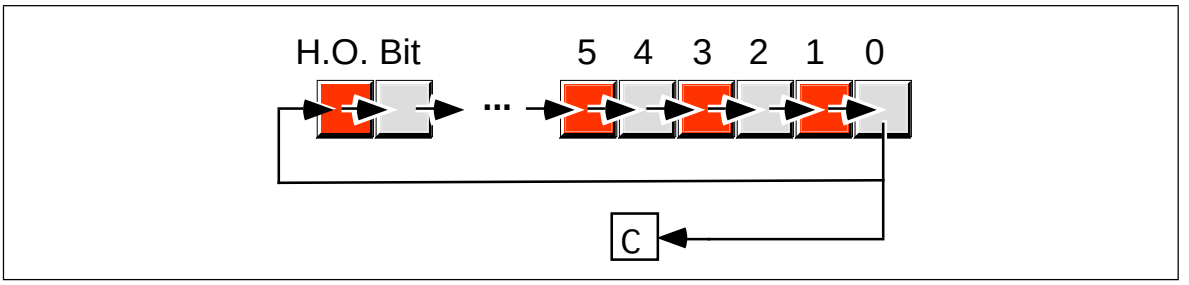


Figure 6.11 Rotate Right Operation

The ror instruction sets the flags identically to rcr. Other than the source of the bit shifted into the H.O. bit, this instruction behaves exactly like the rcr instruction. **Don't forget the warning about the flags!**

6.6.4 The Bit Operations

Bit twiddling is one of those operations easier done in assembly language than other languages. And no wonder. Most high-level languages shield you from the machine representation of the underlying data types[13]. Instructions like and, or, xor, not, and the shifts and rotates make it possible to test, set, clear, invert, and align bit fields within strings of bits. Even the C++ programming language, famous for its bit manipulation operators, doesn't provide the bit manipulation capabilities of assembly language.

The 80x86 family, particularly the 80386 and later processors, go much farther, though. Besides the standard logical, shift, and rotate instructions, there are instructions to test bits within an operand, to test and set, clear, or invert specific bits in an operand, and to search for set bits. These instructions are

```
test      dest, source
bt        source, index
btc       source, index
btr       source, index
bts       source, index
bsf       dest, source
bsr       dest, source
```

The specific forms are

```
test      reg, reg
test      reg, mem
test      mem, reg                        (*)
test      reg, imm
test      mem, imm
test      eax/ax/al, imm

bt        reg, reg                        (3)
bt        mem, reg                        (3)
bt        reg, imm                        (3)
bt        mem, imm                        (3)

btc uses the same formats as bt.          (3)
btr uses the same formats as bt.          (3)
bts uses the same formats as bt.          (3)

bsf       reg, reg                        (3)
bsr       reg, mem                        (3)

bsr uses the same formats as bsf.         (3)

3- This instruction is only available on 80386 and later processors.
*- This is the same instruction as test reg,mem
```

Note that the bt, btc, btr, bts, bsf, and bsr require 16 or 32 bit operands.

13. Indeed, this is one of the purposes of high level languages, to hide such low-level details.

The bit operations are useful when implementing (monochrome) bit mapped graphic primitive functions and when implementing a set data type using bit maps.

6.6.4.1 TEST

The test instruction logically ands its two operands and sets the flags but does not save the result. Test and and share the same relationship as cmp and sub. Typically, you would use this instruction to see if a bit contains one. Consider the following instruction:

```
test    al, 1
```

This instruction logically ands al with the value one. If bit zero of al contains a one, the result is non-zero and the 80x86 clears the zero flag. If bit zero of al contains zero, then the result is zero and the test operation sets the zero flag. You can test the zero flag after this instruction to decide whether al contained zero or one in bit zero.

The test instruction can also check to see if one or more bits in a register or memory location are non-zero. Consider the following instruction:

```
test    dx, 105h
```

This instruction logically ands dx with the value 105h. This will produce a non-zero result (and, therefore, clear the zero flag) if at least one of bits zero, two, or eight contain a one. They must all be zero to set the zero flag.

The test instruction sets the flags identically to the and instruction:

- It clears the carry flag.
- It clears the overflow flag.
- It sets the zero flag if the result is zero, they clear it otherwise.
- It copies the H.O. bit of the result into the sign flag.
- It sets the parity flag according to the parity (number of one bits) in the L.O. byte of the result.
- It scrambles the auxiliary carry flag.

6.6.4.2 The Bit Test Instructions: BT, BTS, BTR, and BTC

On an 80386 or later processor, you can use the bt instruction (*bit test*) to test a single bit. Its second operand specifies the *bit index* into the first operand. Bt copies the addressed bit into the carry flag. For example, the instruction

```
bt      ax, 12
```

copies bit twelve of ax into the carry flag.

The bt/bts/btr/btc instructions only deal with 16 or 32 bit operands. This is not a limitation of the instruction. After all, if you want to test bit three of the al register, you can just as easily test bit three of the ax register. On the other hand, if the index is larger than the size of a register operand, the result is undefined.

If the first operand is a memory location, the bt instruction tests the bit at the given offset in memory, regardless the value of the index. For example, if bx contains 65 then

```
bt      TestMe, bx
```

will copy bit one of location TestMe+8 into the carry flag. Once again, the size of the operand does not matter. For all intents and purposes, the memory operand is a byte and you can test any bit after that byte with an appropriate index. The actual bit bt tests is at bit position *index mod 8* and at memory offset *effective address + index/8*.

The bts, btr, and btc instructions also copy the addressed bit into the carry flag. However, these instructions also set, reset (clear), or complement (invert) the bit in the first operand after copying it to the carry flag. This provides *test and set*, *test and clear*, and *test and invert* operations necessary for some concurrent algorithms.

The bt, bts, btr, and btc instructions do not affect any flags other than the carry flag.

6.6.4.3 Bit Scanning: BSF and BSR

The bsf (Bit Scan Forward) and bsr (Bit Scan Reverse) instructions search for the first or last set bit in a 16 or 32 bit quantity. The general form of these instructions is

```
bsf     dest, source
bsr     dest, source
```

Bsf locates the first set bit in the source operand, searching from bit zero though the H.O. bit. Bsr locates the first set bit searching from the H.O. bit down to the L.O. bit. If these instructions locate a one, they clear the zero flag and store the bit index (0..31) into the destination operand. If the source operand is zero, these instructions set the zero flag and store an indeterminate value into the destination operand[14].

To scan for the first bit containing zero (rather than one), make a copy of the source operand and invert it (using not), then execute bsf or bsr on the inverted value. The zero flag would be set after this operation if there were no zero bits in the original source value, otherwise the destination operation will contain the position of the first bit containing zero.

6.6.5 The "Set on Condition" Instructions

The *set on condition* (or set*cc*) instructions set a single byte operand (register or memory location) to zero or one depending on the values in the flags register. The general formats for the set*cc* instructions are

```
setcc   reg8
setcc   mem8
```

Set*cc* represents a mnemonic appearing in the following tables. These instructions store a zero into the corresponding operand if the condition is false, they store a one into the eight bit operand if the condition is true.

Table 28: SETcc Instructions That Test Flags

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if no sign	Sign = 0	
SETO	Set if overflow	Ovrflw=1	
SETNO	Set if no overflow	Ovrflw=0	
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

14. On many of the processors, if the source operand is zero the CPU will leave the destination operand unchanged. However, certain versions of the 80486 do scramble the destination operand, so you shouldn't count on it being unchanged if the source operand is zero.

The set*cc* instructions above simply test the flags without any other meaning attached to the operation. You could, for example, use setc to check the carry flag after a shift, rotate, bit test, or arithmetic operation. Likewise, you could use setnz instruction after a test instruction to check the result.

The cmp instruction works synergistically with the set*cc* instructions. Immediately after a cmp operation the processor flags provide information concerning the relative values of those operands. They allow you to see if one operand is less than, equal to, greater than, or any combination of these.

There are two groups of set*cc* instructions that are very useful after a cmp operation. The first group deals with the result of an *unsigned* comparison, the second group deals with the result of a *signed* comparison.

Table 29: SETcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Comments
SETA	Set if above (>)	Carry=0, Zero=0	Same as SETNBE
SETNBE	Set if not below or equal (not <=)	Carry=0, Zero=0	Same as SETA
SETAE	Set if above or equal (>=)	Carry = 0	Same as SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Same as SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Same as SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Same as SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 or Zero = 1	Same as SETNA
SETNA	Set if not above (not >)	Carry = 1 or Zero = 1	Same as SETBE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The corresponding table for signed comparisons is

Table 30: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETG	Set if greater (>)	Sign = Ovrflw or Zero=0	Same as SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	Same as SETG
SETGE	Set if greater than or equal (>=)	Sign = Ovrflw	Same as SETNL
SETNL	Set if not less than (not <)	Sign = Ovrflw	Same as SETGE
SETL	Set if less than (<)	Sign ≠ Ovrflw	Same as SETNGE
SETNGE	Set if not greater or equal (not >=)	Sign ≠ Ovrflw	Same as SETL
SETLE	Set if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	Same as SETNG
SETNG	Set if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	Same as SETLE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The set*cc* instructions are particularly valuable because they can convert the result of a comparison to a boolean value (true/false or 0/1). This is especially important when

translating statements from a high level language like Pascal or C++ into assembly language. The following example shows how to use these instructions in this manner:

```
; Bool := A <= B

                mov     ax, A       ;Assume A and B are signed integers.
                cmp     ax, B
                setle   Bool        ;Bool needs to be a byte variable.
```

Since the set*cc* instructions always produce zero or one, you can use the results with the logical and and or instructions to compute complex boolean values:

```
; Bool := ((A <= B) and (D = E)) or (F <> G)

                mov     ax, A
                cmp     ax, B
                setle   bl
                mov     ax, D
                cmp     ax, E
                sete    bh
                and     bl, bh
                mov     ax, F
                cmp     ax, G
                setne   bh
                or      bl, bh
                mov     Bool, bh
```

For more examples, see “Logical (Boolean) Expressions” on page 467.

The set*cc* instructions always produce an eight bit result since a byte is the smallest operand the 80x86 will operate on. However, you can easily use the shift and rotate instructions to pack eight boolean values in a single byte. The following instructions compare eight different values with zero and copy the “zero flag” from each comparison into corresponding bits of al:

```
                cmp     Val7, 0
                setne   al          ;Put first value in bit #0
                cmp     Val6, 0     ;Test the value for bit #6
                setne   ah          ;Copy zero flag into ah register.
                shr     ah, 1       ;Copy zero flag into carry.
                rcl     al, 1       ;Shift carry into result byte.
                cmp     Val5, 0     ;Test the value for bit #5
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val4, 0     ;Test the value for bit #4
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val3, 0     ;Test the value for bit #3
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val2, 0     ;Test the value for bit #2
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val1, 0     ;Test the value for bit #1
                setne   ah
                shr     ah, 1
                rcl     al, 1
                cmp     Val0, 0     ;Test the value for bit #0
                setne   ah
                shr     ah, 1
                rcl     al, 1

; Now AL contains the zero flags from the eight comparisons.
```

6.7 I/O Instructions

The 80x86 supports two I/O instructions: in and out[15]. They take the forms:

```
in      eax/ax/al, port
in      eax/ax/al, dx
out     port, eax/ax/al
out     dx, eax/ax/al
```

port is a value between 0 and 255.

The 80x86 supports up to 65,536 different I/O ports (requiring a 16 bit I/O address). The port value above, however, is a single byte value. Therefore, you can only directly address the first 256 I/O ports in the 80x86's I/O address space. To address all 65,536 different I/O ports, you must load the address of the desired port (assuming it's above 255) into the dx register and access the port indirectly. The in instruction reads the data at the specified I/O port and copies it into the accumulator. The out instruction writes the value in the accumulator to the specified I/O port.

Please realize that there is nothing magical about the 80x86's in and out instructions. They're simply another form of the mov instruction that accesses a different memory space (the I/O address space) rather than the 80x86's normal 1 Mbyte memory address space.

The in and out instructions do not affect any 80x86 flags.

Examples of the 80x86 I/O instructions:

```
in      al, 60h     ;Read keyboard port

mov     dx, 378h    ;Point at LPT1: data port
in      al, dx      ;Read data from printer port.
inc     ax          ;Bump the ASCII code by one.
out     dx, al      ;Write data in AL to printer port.
```

6.8 String Instructions

The 80x86 supports twelve string instructions:

- movs (move string)
- lods (load string element into the accumulator)
- stos (store accumulator into string element)
- scas (Scan string and check for match against the value in the accumulator)
- cmps (compare two strings)
- ins (input a string from an I/O port)
- outs (output a string to an I/O port
- rep (repeat a string operation)
- repz (repeat while zero)
- repe (repeat while equal)
- repnz (repeat while not zero)
- repne (repeat while not equal)

You can use the movs, stos, scas, cmps, ins and outs instructions to manipulate a single element (byte, word, or double word) in a string, or to process an entire string. Generally, you would only use the lods instruction to manipulate a single item at a time.

These instructions can operate on strings of bytes, words, or double words. To specify the object size, simply append a *b*, *w*, or *d* to the end of the instruction's mnemonic, i.e., lodsb, movsw, cmpsd, etc. Of course, the double word forms are only available on 80386 and later processors.

15. Actually, the 80286 and later processors support four I/O instructions, you'll get a chance to see the other two in the next section.

The movs and cmps instructions assume that ds:si contains the segmented address of a source string and that es:di contains the segmented address of a destination string. The lods instruction assumes that ds:si points at a source string, the accumulator (al/ax/eax) is the destination location. The scas and stos instructions assume that es:di points at a destination string and the accumulator contains the source value.

The movs instruction moves one string element (byte, word, or dword) from memory location ds:si to es:di. After moving the data, the instruction increments or decrements si and di by one, two, or four if processing bytes, words, or dwords, respectively. The CPU increments these registers if the direction flag is clear, the CPU decrements them if the direction flag is set.

The movs instruction can move blocks of data around in memory. You can use it to move strings, arrays, and other multi-byte data structures.

```
movs{b,w,d}:    es:[di] := ds:[si]
                if direction_flag = 0 then
                        si := si + size;
                        di := di + size;
                else
                        si := si - size;
                        di := di - size;
                endif;
```

Note: *size* is one for bytes, two for words, and four for dwords.

The cmps instruction compares the byte, word, or dword at location ds:si to es:di and sets the processor flags accordingly. After the comparison, cmps increments or decrements si and di by one, two, or four depending on the size of the instruction and the status of the direction flag in the flags register.

```
cmps{b,w,d}:    cmp ds:[si], es:[di]
                if direction_flag = 0 then
                        si := si + size;
                        di := di + size;
                else
                        si := si - size;
                        di := di - size;
                endif;
```

The lods instruction moves the byte, word, or dword at ds:si into the al, ax, or eax register. It then increments or decrements the si register by one, two, or four depending on the instruction size and the value of the direction flag. The lods instruction is useful for fetching a sequence of bytes, words, or double words from an array, performing some operation(s) on those values and then processing the next element from the string.

```
lods{b,w,d}:    eax/ax/al := ds:[si]
                if direction_flag = 0 then
                        si := si + size;
                else
                        si := si - size;
                endif;
```

The stos instruction stores al, ax, or eax at the address specified by es:di. Again, di is incremented or decremented according to the size of the instruction and the value of the direction flag. The stos instruction has several uses. Paired with the lods instruction above, you can load (via lods), manipulate, and store string elements. By itself, the stos instruction can quickly store a single value throughout a multi-byte data structure.

```
stos{b,w,d}:    es:[di] := eax/ax/al
                if direction_flag = 0 then
                        di := di + size;
                else
                        di := di - size;
                endif;
```

The scas instruction compares al, ax or eax against the value at location es:di and then adjusts di accordingly. This instruction sets the flags in the processor status register just

like the cmp and cmps instructions. The scas instruction is great for searching for a particular value throughout some multi-byte data structure.

```
scas{b,w,d}:    cmp eax/ax/al, es:[di]
                if direction_flag = 0 then
                        di := di + size;
                else
                        di := di - size;
                endif;
```

The ins instruction inputs a byte, word, or double word from the I/O port specified in the dx register. It then stores the input value at memory location es:di and increments or decrements di appropriately. This instruction is available only on 80286 and later processors.

```
ins{b,w,d}:     es:[di] := port(dx)
                if direction_flag = 0 then
                        di := di + size;
                else
                        di := di - size;
                endif;
```

The outs instruction fetches the byte, word, or double word at address ds:si, increments or decrements si accordingly, and then outputs the value to the port specified in the dx register.

```
outs{b,w,d}:    port(dx) := ds:[si]
                if direction_flag = 0 then
                        si := si + size;
                else
                        si := si - size;
                endif;
```

As explained here, the string instructions are useful, but it gets even better! When combined with the rep, repz, repe, repnz, and repne prefixes, a single string instruction can process an entire string. For more information on these prefixes see the chapter on strings.

6.9 Program Flow Control Instructions

The instructions discussed thus far execute sequentially; that is, the CPU executes each instruction in the sequence it appears in your program. To write real programs requires several control structures, not just the sequence. Examples include the if statement, loops, and subroutine invocation (a call). Since compilers reduce all other languages to assembly language, it should come as no surprise that assembly language supports the instructions necessary to implement these control structures. 80x86 program control instructions belong to three groups: unconditional transfers, conditional transfers, and subroutine call and return instructions. The following sections describe these instructions:

6.9.1 Unconditional Jumps

The jmp (jump) instruction unconditionally transfers control to another point in the program. There are six forms of this instruction: an intersegment/direct jump, two intrasegment/direct jumps, an intersegment/indirect jump, and two intrasegment/indirect jumps. Intrasegment jumps are always between statements in the same code segment. Intersegment jumps can transfer control to a statement in a different code segment.

These instructions generally use the same syntax, it is

```
jmp     target
```

The assembler differentiates them by their operands:

```
jmp     disp8       ;direct intrasegment, 8 bit displacement.
jmp     disp16      ;direct intrasegment, 16 bit displacement.
jmp     adrs32      ;direct intersegment, 32 bit segmented address.
```

```
        jmp     mem16       ;indirect intrasegment, 16 bit memory operand.
        jmp     reg16       ;register indirect intrasegment.
        jmp     mem32       ;indirect intersegment, 32 bit memory operand.
```

Intersegment is a synonym for *far*, intrasegment is a synonym for *near*.

The two direct intrasegment jumps differ only in their length. The first form consists of an opcode and a single byte displacement. The CPU sign extends this displacement to 16 bits and adds it to the ip register. This instruction can branch to a location -128..+127 from the beginning of the next instruction following it (i.e., -126..+129 bytes around the current instruction).

The second form of the intrasegment jump is three bytes long with a two byte displacement. This instruction allows an effective range of -32,768..+32,767 bytes and can transfer control to anywhere in the current code segment. The CPU simply adds the two byte displacement to the ip register.

These first two jumps use a *relative* addressing scheme. The offset encoded as part of the opcode byte is *not* the target address in the current code segment, but the distance to the target address. Fortunately, MASM will compute the distance for you automatically, so you do not have to compute this displacement value yourself. In many respects, these instructions are really nothing more than add ip, disp instructions.

The direct intersegment jump is five bytes long, the last four bytes containing a segmented address (the offset in the second and third bytes, the segment in the fourth and fifth bytes). This instruction copies the offset into the ip register and the segment into the cs register. Execution of the next instruction continues at the new address in cs:ip. Unlike the previous two jumps, the address following the opcode is the absolute memory address of the target instruction; this version does not use relative addressing. This instruction loads cs:ip with a 32 bit immediate value.

For the three direct jumps described above, you normally specify the target address using a *statement label*. A statement label is usually an identifier followed by a colon, usually on the same line as an executable machine instruction. The assembler determines the offset of the statement after the label and automatically computes the distance from the jump instruction to the statement label. Therefore, you do not have to worry about computing displacements manually. For example, the following short little loop continuously reads the parallel printer data port and inverts the L.O. bit. This produces a *square wave* electrical signal on one of the printer port output lines:

```
                mov     dx, 378h    ;Parallel printer port address.
LoopForever:    in      al, dx      ;Read character from input port.
                xor     al, 1       ;Invert the L.O. bit.
                out     dx, al      ;Output data back to port.
                jmp     LoopForever ;Repeat forever.
```

The fourth form of the unconditional jump instruction is the indirect intrasegment jump instruction. It requires a 16 bit memory operand. This form transfers control to the address within the offset given by the two bytes of the memory operand. For example,

```
WordVar         word    TargetAddress
                  .
                  .
                  .
                jmp     WordVar
```

transfers control to the address specified by the value in the 16 bit memory location WordVar. This does *not* jump to the statement at address WordVar, it jumps to the statement at the address held in the WordVar variable. Note that this form of the jmp instruction is roughly equivalent to:

```
                mov     ip, WordVar
```

Although the example above uses a single word variable containing the indirect address, you can use *any* valid memory address mode, not just the displacement only addressing mode. You can use memory indirect addressing modes like the following:

```
                jmp     DispOnly    ;Word variable
```

```
                jmp     Disp[bx]    ;Disp is an array of words
                jmp     Disp[bx][si]
                jmp     [bx][16]
                etc.
```

Consider the indexed addressing mode above for a moment (disp[bx]). This addressing mode fetches the word from location disp+bx and copies this value to the ip register; this lets you create an array of pointers and jump to a specified pointer using an array index. Consider the following example:

```
AdrsArray       word    stmt1, stmt2, stmt3, stmt4
                 .
                 .
                 .
                mov     bx, I       ;I is in the range 0..3
                add     bx, bx      ;Index into an array of words.
                jmp     AdrsArray[bx];Jump to stmt1, stmt2, etc., depending
                                    ; on the value of I.
```

The important thing to remember is that the *near indirect* jump fetches a word from memory and copies it into the ip register; it does *not* jump to the memory location specified, it jumps indirectly through the 16 bit pointer at the specified memory location.

The fifth jmp instruction transfers control to the offset given in a 16 bit general purpose register. Note that you can use *any* general purpose register, not just bx, si, di, or bp. An instruction of the form

```
                jmp     ax
```

is roughly equivalent to

```
                mov     ip, ax
```

Note that the previous two forms (register or memory indirect) are really the same instruction. The mod and r/m fields of a mod-reg-r/m byte specify a register or memory indirect address. See Appendix D for the details.

The sixth form of the jmp instruction, the indirect intersegment jump, has a memory operand that contains a double word pointer. The CPU copies the double word at that address into the cs:ip register pair. For example,

```
FarPointer      dword   TargetAddress
                 .
                 .
                 .
                jmp     FarPointer
```

transfers control to the segmented address specified by the four bytes at address FarPointer. This instruction is semantically identical to the (mythical) instruction

```
                lcs ip, FarPointer    ;load cs, ip from FarPointer
```

As for the near indirect jump described earlier, this *far indirect* jump lets you specify any arbitrary (valid) memory addressing mode. You are not limited to the displacement only addressing mode the example above uses.

MASM uses a near indirect or far indirect addressing mode depending upon the type of the memory location you specify. If the variable you specify is a word variable, MASM will automatically generate a near indirect jump; if the variable is a dword, MASM emits the opcode for a far indirect jump. Some forms of memory addressing, unfortunately, do not intrinsically specify a size. For example, [bx] is definitely a memory operand, but does bx point at a word variable or a double word variable? It *could* point at either. Therefore, MASM will reject a statement of the form:

```
                jmp     [bx]
```

MASM cannot tell whether this should be a near indirect or far indirect jump. To resolve the ambiguity, you will need to use a *type coercion operator*. Chapter Eight will fully

16. Technically, this is syntactically incorrect because MASM cannot figure out the size of the memory operand. Read on for the details.

describe type coercion operators, for now, just use one of the following two instructions for a near or far jump, respectively:

```
jmp     word ptr [bx]
jmp     dword ptr [bx]
```

The register indirect addressing modes are not the only ones that could be type ambiguous. You could also run into this problem with indexed and base plus index addressing modes:

```
jmp     word ptr 5[bx]
jmp     dword ptr 9[bx][si]
```

For more information on the type coercion operators, see Chapter Eight.

In theory, you could use the indirect jump instructions and the set*cc* instructions to *conditionally* transfer control to some given location. For example, the following code transfers control to iftrue if word variable X is equal to word variable Y. It transfers control to iffalse, otherwise.

```
JmpTbl          word    iffalse, iftrue
                 .
                 .
                 .
                mov     ax, X
                cmp     ax, Y
                sete    bl
                movzx   ebx, bl
                jmp     JmpTbl[ebx*2]
```

As you will soon see, there is a *much* better way to do this using the conditional jump instructions.

6.9.2 The CALL and RET Instructions

The call and ret instructions handle subroutine calls and returns. There are five different call instructions and six different forms of the return instruction:

```
call    disp16      ;direct intrasegment, 16 bit relative.
call    adrs32      ;direct intersegment, 32 bit segmented address.
call    mem16       ;indirect intrasegment, 16 bit memory pointer.
call    reg16       ;indirect intrasegment, 16 bit register pointer.
call    mem32       ;indirect intersegment, 32 bit memory pointer.

ret                 ;near or far return
retn                ;near return
retf                ;far return
ret     disp        ;near or far return and pop
retn    disp        ;near return and pop
retf    disp        ;far return and pop
```

The call instructions take the same forms as the jmp instructions except there is no short (two byte) intrasegment call.

The far call instruction does the following:

- It pushes the cs register onto the stack.
- It pushes the 16 bit offset of the next instruction following the call onto the stack.
- It copies the 32 bit effective address into the cs:ip registers. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
- Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The near call instruction does the following:

- It pushes the 16 bit offset of the next instruction following the call onto the stack.
- It copies the 16 bit effective address into the ip register. Since the call instruction allows the same addressing modes as jmp, call can obtain the target address using a relative, memory, or register addressing mode.
- Execution continues at the first instruction of the subroutine. This first instruction is the opcode at the target address computed in the previous step.

The call $disp_{16}$ instruction uses relative addressing. You can compute the effective address of the target by adding this 16 bit displacement with the return address (like the relative jmp instructions, the displacement is the distance from the instruction *following* the call to the target address).

The call $adrs_{32}$ instruction uses the direct addressing mode. A 32 bit segmented address immediately follows the call opcode. This form of the call instruction copies that value directly into the cs:ip register pair. In many respects, this is equivalent to the immediate addressing mode since the value this instruction copies into the cs:ip register pair immediately follows the instruction.

Call mem_{16} uses the memory indirect addressing mode. Like the jmp instruction, this form of the call instruction fetches the word at the specified memory location and uses that word's value as the target address. Remember, you can use *any* memory addressing mode with this instruction. The displacement-only addressing mode is the most common form, but the others are just as valid:

```
call      CallTbl[bx]           ;Index into an array of pointers.
call      word ptr [bx]         ;BX points at word to use.
call      WordTbl[bx][si]       ; etc.
```

Note that the selection of addressing mode only affects the effective address computation for the target subroutine. These call instructions still push the offset of the next instruction following the call onto the stack. Since these are *near* calls (they obtain their target address from a 16 bit memory location), they all push a 16 bit return address onto the stack.

Call reg_{16} works just like the memory indirect call above, except it uses the 16 bit value in a register for the target address. This instruction is really the same instruction as the call mem_{16} instruction. Both forms specify their effective address using a mod-reg-r/m byte. For the call reg_{16} form, the mod bits contain 11b so the r/m field specifies a register rather than a memory addressing mode. Of course, this instruction also pushes the 16 bit offset of the next instruction onto the stack as the return address.

The call mem_{32} instruction is a far indirect call. The memory address specified by this instruction must be a double word value. This form of the call instruction fetches the 32 bit segmented address at the computed effective address and copies this double word value into the cs:ip register pair. This instruction also copies the 32 bit segmented address of the next instruction onto the stack (it pushes the segment value first and the offset portion second). Like the call mem_{16} instruction, you can use any valid memory addressing mode with this instruction:

```
call      DWordVar
call      DwordTbl[bx]
call      dword ptr [bx]
etc.
```

It is relatively easy to synthesize the call instruction using two or three other 80x86 instructions. You could create the equivalent of a near call using a push and a jmp instruction:

```
push      <offset of instruction after jmp>
jmp       subroutine
```

A far call would be similar, you'd need to add a push cs instruction before the two instructions above to push a far return address on the stack.

The ret (return) instruction returns control to the caller of a subroutine. It does so by popping the return address off the stack and transferring control to the instruction at this

return address. Intrasegment (near) returns pop a 16 bit return address off the stack into the ip register. An intersegment (far) return pops a 16 bit offset into the ip register and then a 16 bit segment value into the cs register. These instructions are effectively equal to the following:

```
retn:           pop     ip
retf:           popd    cs:ip
```

Clearly, you must match a near subroutine call with a near return and a far subroutine call with a corresponding far return. If you mix near calls with far returns or vice versa, you will leave the stack in an inconsistent state and you probably will *not* return to the proper instruction after the call. Of course, another important issue when using the call and ret instructions is that you must make sure your subroutine doesn't push something onto the stack and then fail to pop it off before trying to return to the caller. Stack problems are a major cause of errors in assembly language subroutines. Consider the following code:

```
Subroutine:     push    ax
                push    bx
                 .
                 .
                 .
                pop     bx
                ret
                 .
                 .
                 .
                call    Subroutine
```

The call instruction pushes the return address onto the stack and then transfers control to the first instruction of subroutine. The first two push instructions push the ax and bx registers onto the stack, presumably in order to preserve their value because subroutine modifies them. Unfortunately, a programming error exists in the code above, subroutine only pops bx from the stack, it fails to pop ax as well. This means that when subroutine tries to return to the caller, the value of ax rather than the return address is sitting on the top of the stack. Therefore, this subroutine returns control to the address specified by the initial value of the ax register rather than to the true return address. Since there are 65,536 different values ax can have, there is a $^{1}/_{65,536}^{th}$ of a chance that your code will return to the real return address. The odds are not in your favor! Most likely, code like this will hang up the machine. Moral of the story – always make sure the return address is sitting on the stack before executing the return instruction.

Like the call instruction, it is very easy to simulate the ret instruction using two 80x86 instructions. All you need to do is pop the return address off the stack and then copy it into the ip register. For near returns, this is a very simple operation, just pop the near return address off the stack and then jump indirectly through that register:

```
                pop     ax
                jmp     ax
```

Simulating a far return is a little more difficult because you must load cs:ip in a single operation. The only instruction that does this (other than a far return) is the jmp mem_{32} instruction. See the exercises at the end of this chapter for more details.

There are two other forms of the ret instruction. They are identical to those above except a 16 bit displacement follows their opcodes. The CPU adds this value to the stack pointer immediately after popping the return address from the stack. This mechanism removes parameters pushed onto the stack before returning to the caller. See "Passing Parameters on the Stack" on page 581 for more details.

The assembler allows you to type ret without the "f" or "n" suffix. If you do so, the assembler will figure out whether it should generate a near return or a far return. See the chapter on procedures and functions for details on this.

6.9.3 The INT, INTO, BOUND, and IRET Instructions

The int (for software interrupt) instruction is a very special form of a call instruction. Whereas the call instruction calls subroutines within your program, the int instruction calls system routines and other special subroutines. The major difference between *interrupt service routines* and standard procedures is that you can have any number of different procedures in an assembly language program, while the *system* supports a maximum of 256 different interrupt service routines. A program calls a subroutine by specifying the *address* of that subroutine; it calls an interrupt service routine by specifying the *interrupt number* for that particular interrupt service routine. This chapter will only describe how to *call* an interrupt service routine using the int, into, and bound instructions, and how to return from an interrupt service routine using the iret instruction.

There are four different forms of the int instruction. The first form is

```
int       nn
```

(where "nn" is a value between 0 and 255). It allows you to call one of 256 different interrupt routines. This form of the int instruction is two bytes long. The first byte is the int opcode. The second byte is immediate data containing the interrupt number.

Although you can use the int instruction to call procedures (interrupt service routines) you've written, the primary purpose of this instruction is to make a *system call*. A system call is a subroutine call to a procedure provided by the system, such as a DOS , PC-BIOS[17], mouse, or some other piece of software resident in the machine before your program began execution. Since you always refer to a specific system call by its interrupt number, rather than its address, your program does not need to know the actual address of the subroutine in memory. The int instruction provides *dynamic linking* to your program. The CPU determines the actual address of an interrupt service routine at run time by looking up the address in an *interrupt vector table*. This allows the authors of such system routines to change their code (including the entry point) without fear of breaking any older programs that call their interrupt service routines. As long as the system call uses the same interrupt number, the CPU will automatically call the interrupt service routine at its new address.

The only problem with the int instruction is that it supports only 256 different interrupt service routines. MS-DOS alone supports well over 100 different calls. BIOS and other system utilities provide thousands more. This is above and beyond all the interrupts reserved by Intel for hardware interrupts and traps. The common solution most of the system calls use is to employ a *single* interrupt number for a given class of calls and then pass a *function number* in one of the 80x86 registers (typically the ah register). For example, MS-DOS uses only a single interrupt number, 21h. To choose a particular DOS function, you load a DOS *function code* into the ah register before executing the int 21h instruction. For example, to terminate a program and return control to MS-DOS, you would normally load ah with 4Ch and call DOS with the int 21h instruction:

```
mov       ah, 4ch       ;DOS terminate opcode.
int       21h           ;DOS call
```

The BIOS keyboard interrupt is another good example. Interrupt 16h is responsible for testing the keyboard and reading data from the keyboard. This BIOS routine provides several calls to read a character and scan code from the keyboard, see if any keys are available in the system type ahead buffer, check the status of the keyboard modifier flags, and so on. To choose a particular operation, you load the function number into the ah register before executing int 16h. The following table lists the possible functions:

17. BIOS stands for Basic Input/Output System.

Table 31: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0		al- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows: bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down
3	al = 5 bh = 0, 1, 2, 3 for 1/4, 1/2, 3/4, or 1 second delay bl= 0..1Fh for 30/sec to 2/sec.		Set auto repeat rate. The bh register contains the amount of time to wait before starting the autorepeat operation, the bl register contains the autorepeat rate.
5	ch = scan code cl = ASCII code		Store keycode in buffer. This function stores the value in the cx register at the end of the type ahead buffer. Note that the scan code in ch doesn't have to correspond to the ASCII code appearing in cl. This routine will simply insert the data you provide into the system type ahead buffer.
10h		al- ASCII character ah- scan code	Read extended character. Like ah=0 call, except this one passes all key codes, the ah=0 call throws away codes that are not PC/XT compatible.
11h		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Like the ah=01h call except this one does not throw away keycodes that are not PC/XT compatible (i.e., the extra keys found on the 101 key keyboard).

Table 31: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
12h		al- shift flags ah- extended shift flags	Returns the current status of the shift flags in ax. The shift flags are defined as follows: bit 15: SysReq key pressed bit 14: Capslock key currently down bit 13: Numlock key currently down bit 12: Scroll lock key currently down bit 11: Right alt key is down bit 10:Right ctrl key is down bit 9: Left alt key is down bit 8: Left ctrl key is down bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Either alt key is down (some machines, left only) bit 2: Either ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

For example, to read a character from the system type ahead buffer, leaving the ASCII code in al, you could use the following code:

```
mov       ah, 0                 ;Wait for key available, and then
int       16h                   ; read that key.
mov       character, al         ;Save character read.
```

Likewise, if you wanted to test the type ahead buffer to see if a key is available, *without reading that keystroke*, you could use the following code:

```
mov       ah, 1                 ;Test to see if key is available.
int       16h                   ;Sets the zero flag if a key is not
                                ; available.
```

For more information about the PC-BIOS and MS-DOS, see “MS-DOS, PC-BIOS, and File I/O” on page 699.

The second form of the int instruction is a special case:

```
int       3
```

Int 3 is a special form of the interrupt instruction that is only one byte long. CodeView and other debuggers use it as a software breakpoint instruction. Whenever you set a breakpoint on an instruction in your program, the debugger will typically replace the first byte of the instruction’s opcode with an int 3 instruction. When your program executes the int 3 instruction, this makes a “system call” to the debugger so the debugger can regain control of the CPU. When this happens, the debugger will replace the int 3 instruction with the original opcode.

While operating inside a debugger, you can explicitly use the int 3 instruction to stop program executing and return control to the debugger. *This is not, however, the normal way to terminate a program*. If you attempt to execute an int 3 instruction while running under DOS, rather than under the control of a debugger program, you will likely crash the system.

The third form of the int instruction is into. Into will cause a software breakpoint if the 80x86 overflow flag is set. You can use this instruction to quickly test for arithmetic overflow after executing an arithmetic instruction. Semantically, this instruction is equivalent to

```
if overflow = 1 then int 4
```

You should *not* use this instruction unless you've supplied a corresponding trap handler (interrupt service routine). Doing so would probably crash the system. .

The fourth software interrupt, provided by 80286 and later processors, is the bound instruction. This instruction takes the form

```
bound     reg, mem
```

and executes the following algorithm:

```
if (reg < [mem]) or (reg > [mem+sizeof(reg)]) then int 5
```

[mem] denotes the contents of the memory location mem and sizeof(reg) is two or four depending on whether the register is 16 or 32 bits wide. The memory operand must be twice the size of the register operand. The bound instruction compares the values using a *signed* integer comparison.

Intel's designers added the bound instruction to allow a quick check of the range of a value in a register. This is useful in Pascal, for example, which checking array bounds validity and when checking to see if a subrange integer is within an allowable range. There are two problems with this instruction, however. On 80486 and Pentium/586 processors, the bound instruction is generally slower than the sequence of instructions it would replace[18]:

```
cmp       reg, LowerBound
jl        OutOfBounds
cmp       reg, UpperBound
jg        OutOfBounds
```

On the 80486 and Pentium/586 chips, the sequence above only requires four clock cycles assuming you can use the immediate addressing mode and the branches are not taken[19]; the bound instruction requires 7-8 clock cycles under similar circumstances and also assuming the memory operands are in the cache.

A second problem with the bound instruction is that it executes an int 5 if the specified register is out of range. IBM, in their infinite wisdom, decided to use the int 5 interrupt handler routine to print the screen. Therefore, if you execute a bound instruction and the value is out of range, the system will, by default, print a copy of the screen to the printer. If you replace the default int 5 handler with one of your own, pressing the PrtSc key will transfer control to your bound instruction handler. Although there are ways around this problem, most people don't bother since the bound instruction is so slow.

Whatever int instruction you execute, the following sequence of events follows:

- The 80x86 pushes the flags register onto the stack;
- The 80x86 pushes cs and then ip onto the stack;
- The 80x86 uses the interrupt number (into is interrupt #4, bound is interrupt #5) times four as an index into the interrupt vector table and copies the double word at that point in the table into cs:ip.

The int instructions vary from a call in two major ways. First, call instructions vary in length from two to six bytes long, whereas int instructions are generally two bytes long (int 3, into, and bound are the exceptions). Second, and most important, the int instruction pushes the flags and the return address onto the stack while the call instruction pushes only the return address. Note also that the int instructions always push a far return address (i.e., a cs value and an offset within the code segment), only the far call pushes this double word return address.

Since int pushes the flags onto the stack you must use a special return instruction, iret (interrupt return), to return from a routine called via the int instructions. If you return from an interrupt procedure using the ret instruction, the flags will be left on the stack upon returning to the caller. The iret instruction is equivalent to the two instruction sequence: ret, popf (assuming, of course, that you execute popf before returning control to the address pointed at by the double word on the top of the stack).

18. The next section describes the jg and jl instructions.

19. In general, one would hope that having a bounds violation is very rare.

The int instructions clear the trace (T) flag in the flags register. They do not affect any other flags. The iret instruction, by its very nature, can affect all the flags since it pops the flags from the stack.

6.9.4 The Conditional Jump Instructions

Although the jmp, call, and ret instructions provide transfer of control, they do not allow you to make any serious decisions. The 80x86's conditional jump instructions handle this task. The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the if..then statement.

The conditional jumps test one or more flags in the flags register to see if they match some particular pattern (just like the set*cc* instructions). If the pattern matches, control transfers to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some instructions, for example, test the conditions of the sign, carry, overflow, and zero flags. For example, after the execution of a shift left instruction, you could test the carry flag to determine if it shifted a one out of the H.O. bit of its operand. Likewise, you could test the condition of the zero flag after a test instruction to see if any specified bits were one. Most of the time, however, you will probably execute a conditional jump after a cmp instruction. The cmp instruction sets the flags so that you can test for less than, greater than, equality, etc.

Note: Intel's documentation defines various synonyms or instruction aliases for many conditional jump instructions. The following tables list all the aliases for a particular instruction. These tables also list out the opposite branches. You'll soon see the purpose of the opposite branches.

Table 32: Jcc Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrflw=1		JNO
JNO	Jump if no Ovrflw	Ovrflw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

Table 33: Jcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposite
JA	Jump if above (>)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Table 34: Jcc Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrflw	JNL	JGE
JNL	Jump if not less than (not <)	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrflw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

On the 80286 and earlier, these instructions are all two bytes long. The first byte is a one byte opcode followed by a one byte displacement. Although this leads to very compact instructions, a single byte displacement only allows a range of ±128 bytes. There is a simple trick you can use to overcome this limitation on these earlier processors:

- Whatever jump you're using, switch to its opposite form. (given in the tables above).
- Once you've selected the opposite branch, use it to jump over a jmp instruction whose target address is the original target address.

For example, to convert:

```
jc      Target
```

to the long form, use the following sequence of instructions:

```
                jnc     SkipJmp
                jmp     Target
SkipJmp:
```

If the carry flag is clear (NC=no carry), then control transfers to label SkipJmp, at the same point you'd be if you were using the jc instruction above. If the carry flag is set when encountering this sequence, control will fall through the jnc instruction to the jmp instruction that will transfer control to Target. Since the jmp instruction allows 16 bit displacement and far operands, you can jump anywhere in the memory using this trick.

One brief comment about the "opposites" column is in order. As mentioned above, when you need to manually extend a branch from ±128 you should choose the opposite branch to branch around a jump to the target location. As you can see in the "aliases" column above, many conditional jump instructions have aliases. This means that there will be aliases for the opposite jumps as well. *Do not use any aliases when extending branches that are out of range*. With only two exceptions, a very simple rule completely describes how to generate an opposite branch:

- If the second letter of the j*cc* instruction is *not* an "n", insert an "n" after the "j". E.g., je becomes jne and jl becomes jnl.
- If the second letter of the j*cc* instruction *is* an "n", then remove that "n" from the instruction. E.g., jng becomes jg, jne becomes je.

The two exceptions to this rule are jpe (jump parity even) and jpo (jump parity odd). These exceptions cause few problems because (a) you'll hardly ever need to test the parity flag, and (b) you can use the aliases jp and jnp synonyms for jpe and jpo. The "N/No N" rule applies to jp and jnp.

Though you *know* that jge is the opposite of jl, get in the habit of using jnl rather than jge. It's too easy in an important situation to start thinking "greater is the opposite of less" and substitute jg instead. You can avoid this confusion by always using the "N/No N" rule.

MASM 6.x and many other modern 80x86 assemblers will automatically convert out of range branches to this sequence for you. There is an option that will allow you to disable this feature. For performance critical code that runs on 80286 and earlier processors, you may want to disable this feature so you can fix the branches yourself. The reason is quite simple, this simple fix always wipes out the pipeline no matter which condition is true since the CPU jumps in either case. One thing nice about conditional jumps is that you do not flush the pipeline or the prefetch queue if you do not take the branch. If one condition is true far more often than the other, you might want to use the conditional jump to transfer control to a jmp nearby, so you can continue to fall through as before. For example, if you have a je target instruction and target is out of range, you could convert it to the following code:

```
                je      GotoTarget
                 .
                 .
                 .
GotoTarget:     jmp     Target
```

Although a branch to target now requires executing *two* jumps, this is much more efficient than the standard conversion if the zero flag is normally clear when executing the je instruction.

The 80386 and later processor provide an extended form of the conditional jump that is four bytes long, with the last two bytes containing a 16 bit displacement. These conditional jumps can transfer control anywhere within the current code segment. Therefore, there is no need to worry about manually extending the range of the jump. If you've told MASM you're using an 80386 or later processor, it will automatically choose the two byte or four byte form, as necessary. See Chapter Eight to learn how to tell MASM you're using an 80386 or later processor.

The 80x86 conditional jump instruction give you the ability to split program flow into one of two paths depending upon some logical condition. Suppose you want to increment the ax register if bx is or equal to cx. You can accomplish this with the following code:

```
                cmp     bx, cx
                jne     SkipStmts
                inc     ax
SkipStmts:
```

The trick is to use the *opposite* branch to skip over the instructions you want to execute if the condition is true. Always use the "opposite branch (N/no N)" rule given earlier to select the opposite branch. You can make the same mistake choosing an opposite branch here as you could when extending out of range jumps.

You can also use the conditional jump instructions to synthesize loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the Enter key (carriage return):

```
                mov     di, 0
ReadLnLoop:     mov     ah, 0       ;INT 16h read key opcode.
                int     16h
                mov     Input[di], al
                inc     di
                cmp     al, 0dh     ;Carriage return ASCII code.
                jne     ReadLnLoop
                mov     Input[di-1],0;Replace carriage return with zero.
```

For more information concerning the use of the conditional jumps to synthesize IF statements, loops, and other control structures, see "Control Structures" on page 521.

Like the set*cc* instructions, the conditional jump instructions come in two basic categories – those that test specific process flag values (e.g., jz, jc, jno) and those that test some condition (less than, greater than, etc.). When testing a condition, the conditional jump instructions almost always follow a cmp instruction. The cmp instruction sets the flags so you can use a ja, jae, jb, jbe, je, or jne instruction to test for unsigned less than, less than or equal, equality, inequality, greater than, or greater than or equal. Simultaneously, the cmp instruction sets the flags so you can also do a signed comparison using the jl, jle, je, jne, jg, and jge instructions.

The conditional jump instructions only test flags, they do not affect any of the 80x86 flags.

6.9.5 The JCXZ/JECXZ Instructions

The jcxz (jump if cx is zero) instruction branches to the target address if cx contains zero. Although you can use it anytime you need to see if cx contains zero, you would normally use it before a loop you've constructed with the loop instructions. The loop instruction can repeat a sequence of operations cx times. If cx equals zero, loop will repeat the operation 65,536 times. You can use jcxz to skip over such a loop when cx is zero.

The jecxz instruction, available only on 80386 and later processors, does essentially the same job as jcxz except it tests the full ecx register. Note that the jcxz instruction only checks cx, even on an 80386 in 32 bit mode.

There are no "opposite" jcxz or jecxz instructions. Therefore, you cannot use "N/No N" rule to extend the jcxz and jecxz instructions. The easiest way to solve this problem is to break the instruction up into two instructions that accomplish the same task:

```
                jcxz    Target
```

becomes

```
                test    cx, cx      ;Sets the zero flag if cx=0
                je      Target
```

Now you can easily extend the je instruction using the techniques from the previous section.

The test instruction above will set the zero flag if and only if cx contains zero. After all, if there are any non-zero bits in cx, logically anding them with themselves will produce a non-zero result. This is an efficient way to see if a 16 or 32 bit register contains zero. In fact, this two instruction sequence is *faster* than the jcxz instruction on the 80486 and later processors. Indeed, Intel recommends the use of this sequence rather than the jcxz instruction if you are concerned with speed. Of course, the jcxz instruction is shorter than the two instruction sequence, but it is not faster. This is a good example of an exception to the rule "shorter is usually faster."

The jcxz instruction does not affect any flags.

6.9.6 The LOOP Instruction

This instruction decrements the cx register and then branches to the target location if the cx register does not contain zero. Since this instruction decrements cx then checks for zero, if cx originally contained zero, any loop you create using the loop instruction will repeat 65,536 times. If you do not want to execute the loop when cx contains zero, use jcxz to skip over the loop.

There is no "opposite" form of the loop instruction, and like the jcxz/jecxz instructions the range is limited to ±128 bytes on all processors. If you want to extend the range of this instruction, you will need to break it down into discrete components:

```
; "loop lbl" becomes:

                dec     cx
                jne     lbl
```

You can easily extend this jne to any distance.

There is no eloop instruction that decrements ecx and branches if not zero (there is a loope instruction, but it does something else entirely). The reason is quite simple. As of the 80386, Intel's designers stopped wholeheartedly supporting the loop instruction. Oh, it's there to ensure compatibility with older code, but it turns out that the dec/jne instructions are actually *faster* on the 32 bit processors. Problems in the decoding of the instruction and the operation of the pipeline are responsible for this strange turn of events.

Although the loop instruction's name suggests that you would normally create loops with it, keep in mind that all it is really doing is decrementing cx and branching to the target address if cx does not contain zero after the decrement. You can use this instruction anywhere you want to decrement cx and then check for a zero result, not just when creating loops. Nonetheless, it is a very convenient instruction to use if you simply want to repeat a sequence of instructions some number of times. For example, the following loop initializes a 256 element array of bytes to the values 1, 2, 3, ...

```
                mov     ecx, 255
ArrayLp:        mov     Array[ecx], cl
                loop    ArrayLp
                mov     Array[0], 0
```

The last instruction is necessary because the loop does not repeat when cx is zero. Therefore, the last element of the array that this loop processes is Array[1], hence the last instruction.

The loop instruction does not affect any flags.

6.9.7 The LOOPE/LOOPZ Instruction

Loope/loopz (loop while equal / zero, they are synonyms for one another) will branch to the target address if cx is not zero and the zero flag is set. This instruction is quite useful

after cmp or cmps instruction, and is marginally faster than the comparable 80386/486 instructions *if you use all the features of this instruction*. However, this instruction plays havoc with the pipeline and superscalar operation of the Pentium so you're probably better off sticking with discrete instructions rather than using this instruction. This instruction does the following:

```
cx := cx - 1
if ZeroFlag = 1 and cx ≠ 0, goto target
```

The loope instruction falls through on one of two conditions. Either the zero flag is clear or the instruction decremented cx to zero. By testing the zero flag after the loop instruction (with a je or jne instruction, for example), you can determine the cause of termination.

This instruction is useful if you need to repeat a loop while some value is equal to another, but there is a maximum number of iterations you want to allow. For example, the following loop scans through an array looking for the first non-zero byte, but it does not scan beyond the end of the array:

```
                mov     cx, 16          ;Max 16 array elements.
                mov     bx, -1          ;Index into the array (note next inc).
SearchLp:       inc     bx              ;Move on to next array element.
                cmp     Array[bx], 0    ;See if this element is zero.
                loope   SearchLp        ;Repeat if it is.
                je      AllZero         ;Jump if all elements were zero.
```

Note that this instruction is not the opposite of loopnz/loopne. If you need to extend this jump beyond ±128 bytes, you will need to synthesize this instruction using discrete instructions. For example, if loope target is out of range, you would need to use an instruction sequence like the following:

```
                jne     quit
                dec     cx
                je      Quit2
                jmp     Target
quit:           dec     cx              ;loope decrements cx, even if ZF=0.
quit2:
```

The loope/loopz instruction does not affect any flags.

6.9.8 The LOOPNE/LOOPNZ Instruction

This instruction is just like the loope/loopz instruction in the previous section except loopne/loopnz (loop while not equal/not zero) repeats while cx is not zero and the zero flag is clear. The algorithm is

```
cx := cx - 1
if ZeroFlag = 0 and cx ≠ 0, goto target
```

You can determine if the loopne instruction terminated because cx was zero or if the zero flag was set by testing the zero flag immediately after the loopne instruction. If the zero flag is clear at that point, the loopne instruction fell through because it decremented cx to zero. Otherwise it fell through because the zero flag was set.

This instruction is *not* the opposite of loope/loopz. If the target address is out of range, you will need to use an instruction sequence like the following:

```
                je      quit
                dec     cx
                je      Quit2
                jmp     Target
quit:           dec     cx              ;loopne decrements cx, even if ZF=1.
quit2:
```

You can use the loopne instruction to repeat some maximum number of times while waiting for some other condition to be true. For example, you could scan through an array until you exhaust the number of array elements or until you find a certain byte using a loop like the following:

```
                mov     cx, 16      ;Maximum # of array elements.
                mov     bx, -1      ;Index into array.
LoopWhlNot0:    inc     bx          ;Move on to next array element.
                cmp     Array[bx],0 ;Does this element contain zero?
                loopne  LoopWhlNot0 ;Quit if it does, or more than 16 bytes.
```

Although the loope/loopz and loopne/loopnz instructions are slower than the individual instruction from which they could be synthesized, there is one main use for these instruction forms where speed is rarely important; indeed, being faster would make them less useful – timeout loops during I/O operations. Suppose bit #7 of input port 379h contains a one if the device is busy and contains a zero if the device is not busy. If you want to output data to the port, you *could* use code like the following:

```
                mov     dx, 379h
WaitNotBusy:    in      al, dx      ;Get port
                test    al, 80h     ;See if bit #7 is one
                jne     WaitNotBusy ;Wait for "not busy"
```

The only problem with this loop is that it is conceivable that it would loop forever. In a real system, a cable could come unplugged, someone could shut off the peripheral device, and any number of other things could go wrong that would hang up the system. Robust programs usually apply a *timeout* to a loop like this. If the device fails to become busy within some specified amount of time, then the loop exits and raises an error condition. The following code will accomplish this:

```
                mov     dx, 379h    ;Input port address
                mov     cx, 0       ;Loop 65,536 times and then quit.
WaitNotBusy:    in      al, dx      ;Get data at port.
                test    al, 80h     ;See if busy
                loopne  WaitNotBusy ;Repeat if busy and no time out.
                jne     TimedOut    ;Branch if CX=0 because we timed out.
```

You could use the loope/loopz instruction if the bit were zero rather than one.

The loopne/loopnz instruction does not affect any flags.

6.10 Miscellaneous Instructions

There are various miscellaneous instructions on the 80x86 that don't fall into any category above. Generally these are instructions that manipulate individual flags, provide special processor services, or handle privileged mode operations.

There are several instructions that directly manipulate flags in the 80x86 flags register. They are

- clc Clears the carry flag
- stc Sets the carry flag
- cmc Complements the carry flag
- cld Clears the direction flag
- std Sets the direction flag
- cli Clears the interrupt enable/disable flag
- sti Sets the interrupt enable/disable flag

Note: you should be careful when using the cli instruction in your programs. Improper use could lock up your machine until you cycle the power.

The nop instruction doesn't do anything except waste a few processor cycles and take up a byte of memory. Programmers often use it as a place holder or a debugging aid. As it turns out, this isn't a unique instruction, it's just a synonym for the xchg ax, ax instruction.

The hlt instruction halts the processor until a reset, non-maskable interrupt, or other interrupt (assuming interrupts are enabled) comes along. Generally, you shouldn't use this instruction on the IBM PC unless you really know what you are doing. *This instruction is not equivalent to the x86 halt instruction. Do not use it to stop your programs.*

The 80x86 provides another prefix instruction, lock, that, like the rep instruction, affects the following instruction. However, this instruction has little meaning on most PC systems. Its purpose is to coordinate systems that have multiple CPUs. As systems become available with multiple processors, this prefix *may* finally become valuable[20]. You need not be too concerned about this here.

The Pentium provides two additional instructions of interest to real-mode DOS programmers. These instructions are cpuid and rdtsc. If you load eax with zero and execute the cpuid instruction, the Pentium (and later processors) will return the maximum value cpuid allows as a parameter in eax. For the Pentium, this value is one. If you load the eax register with one and execute the cpuid instruction, the Pentium will return CPU identification information in eax. Since this instruction is of little value until Intel produces several additional chips in the family, there is no need to consider it further, here.

The second Pentium instruction of interest is the rdtsc (read time stamp counter) instruction. The Pentium maintains a 64 bit counter that counts clock cycles starting at reset. The rdtsc instruction copies the current counter value into the edx:eax register pair. You can use this instruction to accurately time sequences of code.

Besides the instructions presented thus far, the 80286 and later processors provide a set of *protected mode instructions*. This text will not consider those protected most instructions that are useful only to those who are writing operating systems. You would not even use these instructions in your applications when running under a protected mode operating system like Windows, UNIX, or OS/2. These instructions are reserved for the individuals who write such operating systems and drivers for them.

6.11 Sample Programs

The following sample programs demonstrate the use of the various instructions appearing in this chapter.

6.11.1 Simple Arithmetic I

```
; Simple Arithmetic
; This program demonstrates some simple arithmetic instructions.

                .386                    ;So we can use extended registers
                option  segment:use16; and addressing modes.

dseg            segment  para public 'data'

; Some type definitions for the variables we will declare:

uint            typedef  word         ;Unsigned integers.
integer         typedef  sword        ;Signed integers.


; Some variables we can use:

j               integer  ?
k               integer  ?
l               integer  ?

u1              uint     ?
u2              uint     ?
u3              uint     ?
dseg            ends
```

20. There are multiprocessor systems that have multiple Pentium chips installed. However, these systems generally use both CPUs only while running Windows NT, OS/2, or some other operating system that support symmetrical multiprocessing.

```
cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; Initialize our variables:

                mov      j, 3
                mov      k, -2

                mov      u1, 254
                mov      u2, 22

; Compute L := j+k and u3 := u1+u2

                mov      ax, J
                add      ax, K
                mov      L, ax

                mov      ax, u1       ;Note that we use the "ADD"
                add      ax, u2       ; instruction for both signed
                mov      u3, ax       ; and unsigned arithmetic.

; Compute L := j-k and u3 := u1-u2

                mov      ax, J
                sub      ax, K
                mov      L, ax

                mov      ax, u1       ;Note that we use the "SUB"
                sub      ax, u2       ; instruction for both signed
                mov      u3, ax       ; and unsigned arithmetic.

; Compute L := -L

                neg      L

; Compute L := -J

                mov      ax, J        ;Of course, you would only use the
                neg      ax           ; NEG instruction on signed values.
                mov      L, ax

; Compute K := K + 1 using the INC instruction.

                inc      K

; Compute u2 := u2 + 1 using the INC instruction.
; Note that you can use INC for signed and unsigned values.

                inc      u2

; Compute J := J - 1 using the DEC instruction.

                dec      J

; Compute u2 := u2 - 1 using the DEC instruction.
; Note that you can use DEC for signed and unsigned values.

                dec      u2

Quit:           mov      ah, 4ch                ;DOS opcode to quit program.
                int      21h                    ;Call DOS.
Main            endp

cseg            ends
```

```
sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.2 Simple Arithmetic II

```
; Simple Arithmetic
; This program demonstrates some simple arithmetic instructions.

                .386                ;So we can use extended registers
                option   segment:use16; and addressing modes.

dseg            segment  para public 'data'

; Some type definitions for the variables we will declare:

uint            typedef  word       ;Unsigned integers.
integer         typedef  sword      ;Signed integers.


; Some variables we can use:

j               integer  ?
k               integer  ?
l               integer  ?

u1              uint     ?
u2              uint     ?
u3              uint     ?

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; Initialize our variables:

                mov      j, 3
                mov      k, -2

                mov      u1, 254
                mov      u2, 22


; Extended multiplication using 8086 instructions.
;
; Note that there are separate multiply instructions for signed and
; unsigned operands.
;
; L := J * K (ignoring overflow)

                mov      ax, J
                imul     K          ;Computes DX:AX := AX * K
                mov      L, ax      ;Ignore overflow into DX.

; u3 := u1 * u2
```

```
                mov     ax, u1
                mul     u2          ;Computes DX:AX := AX * U2
                mov     u3, ax      ;Ignore overflow in DX.


; Extended division using 8086 instructions.
;
; Like multiplication, there are separate instructions for signed
; and unsigned operands.
;
; It is absolutely imperative that these instruction sequences sign
; extend or zero extend their operands to 32 bits before dividing.
; Failure to do so will may produce a divide error and crash the
; program.
;
; L := J div K

                mov     ax, J
                cwd                 ;*MUST* sign extend AX to DX:AX!
                idiv    K           ;AX := DX:AX/K, DX := DX:AX mod K
                mov     L, ax

; u3 := u1/u2

                mov     ax, u1
                mov     dx, 0       ;Must zero extend AX to DX:AX!
                div     u2          ;AX := DX:AX/u2, DX := DX:AX mod u2
                mov     u3, ax

; Special forms of the IMUL instruction available on 80286, 80386, and
; later processors.  Technically, these instructions operate on signed
; operands only, however, they do work fine for unsigned operands as well.
; Note that these instructions produce a 16-bit result and set the overflow
; flag if overflow occurs.
;
; L := J * 10 (80286 and later only)

                imul    ax, J, 10;AX := J*10
                mov     L, ax

; L := J * K (80386 and later only)

                mov     ax, J
                imul    ax, K
                mov     L, ax



Quit:           mov     ah, 4ch             ;DOS opcode to quit program.
                int     21h                 ;Call DOS.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.3 Logical Operations

```
; Logical Operations
; This program demonstrates the AND, OR, XOR, and NOT instructions
```

```
                .386                    ;So we can use extended registers
                option  segment:use16; and addressing modes.

dseg            segment  para public 'data'


; Some variables we can use:

j               word     0FF00h
k               word     0FFF0h
l               word     ?

c1              byte     'A'
c2              byte     'a'

LowerMask       byte     20h

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; Compute L := J and K (bitwise AND operation):

                mov      ax, J
                and      ax, K
                mov      L, ax

; Compute L := J or K (bitwise OR operation):

                mov      ax, J
                or       ax, K
                mov      L, ax

; Compute L := J xor K (bitwise XOR operation):

                mov      ax, J
                xor      ax, K
                mov      L, ax

; Compute L := not L (bitwise NOT operation):

                not      L

; Compute L := not J (bitwise NOT operation):

                mov      ax, J
                not      ax
                mov      L, ax

; Clear bits 0..3 in J:

                and      J, 0FFF0h

; Set bits 0..3 in K:

                or       K, 0Fh

; Invert bits 4..11 in L:

                xor      L, 0FF0h

; Convert the character in C1 to lower case:
```

```
                mov     al, c1
                or      al, LowerMask
                mov     c1, al

; Convert the character in C2 to upper case:

                mov     al, c2
                and     al, 5Fh     ;Clears bit 5.
                mov     c2, al



Quit:           mov     ah, 4ch             ;DOS opcode to quit program.
                int     21h                 ;Call DOS.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.4 Shift and Rotate Operations

```
; Shift and Rotate Instructions

                .386                ;So we can use extended registers
                option   segment:use16; and addressing modes.

dseg            segment  para public 'data'

; The following structure holds the bit values for an 80x86 mod-reg-r/m byte.

mode            struct
modbits         byte     ?
reg             byte     ?
rm              byte     ?
mode            ends

Adrs1           mode     {11b, 100b, 111b}
modregrm        byte     ?

var1            word     1
var2            word     8000h
var3            word     0FFFFh
var4            word     ?

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; Shifts and rotates directly on memory locations:
;
; var1 := var1 shl 1

                shl      var1, 1
```

```
; var1 := var1 shr 1

                shr     var1, 1

; On 80286 and later processors, you can shift by more than one bit at
; at time:

                shl     var1, 4
                shr     var1, 4

; The arithmetic shift right instruction retains the H.O. bit after each
; shift.  The following SAR instruction sets var2 to 0FFFFh

                sar     var2, 15

; On all processors, you can specify a shift count in the CL register.
; The following instruction restores var2 to 8000h:

                mov     cl, 15
                shl     var2, cl

; You can use the shift and rotate instructions, along with the logical
; instructions, to pack and unpack data.  For example, the following
; instruction sequence extracts bits 10..13 of var3 and leaves
; this value in var4:

                mov     ax, var3
                shr     ax, 10      ;Move bits 10..13 to 0..3.
                and     ax, 0Fh     ;Keep only bits 0..3.
                mov     var4, ax

; You can use the rotate instructions to compute this value somewhat faster
; on older processors like the 80286.

                mov     ax, var3
                rol     ax, 6       ;Six rotates rather than 10 shifts.
                and     ax, 0Fh
                mov     var4, ax

; You can use the shift and OR instructions to easily merge separate fields
; into a single value.  For example, the following code merges the mod, reg,
; and r/m fields (maintained in separate bytes) into a single mod-reg-r/m
; byte:


                mov     al, Adrs1.modbits
                shl     al, 3
                or      al, Adrs1.reg
                shl     al, 3
                or      al, Adrs1.rm
                mov    modregrm, al

; If you've only got and 8086 or 8088 chip, you'd have to use code like the
; following:

                mov     al, Adrs1.modbits;Get mod field
                shl     al, 1
                shl     al, 1
                or      al, Adrs1.reg;Get reg field
                mov     cl, 3
                shl     al, cl                  ;Make room for r/m field.
                or      al, Adrs1.rm            ;Merge in r/m field.
                mov     modregrm, al            ;Save result away.

Quit:           mov     ah, 4ch                 ;DOS opcode to quit program.
                int     21h                     ;Call DOS.
Main            endp

cseg            ends
```

```
sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.5 Bit Operations and SETcc Instructions

```
; Bit Operations and SETcc Instructions

                .386                ;So we can use extended registers
                option   segment:use16; and addressing modes.

dseg            segment  para public 'data'

; Some type definitions for the variables we will declare:

uint            typedef  word       ;Unsigned integers.
integer         typedef  sword      ;Signed integers.


; Some variables we can use:

j               integer  ?
k               integer  ?
u1              uint     2
u2              uint     2
Result          byte     ?

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; Initialize some variables

                mov      j, -2
                mov      k, 2

; The SETcc instructions store a one or zero into their operand if the
; specified condition is true or false, respectively.  The TEST instruction
; logically ANDs its operands and sets the flags accordingly (in particular,
; TEST sets/clears the zero flag if there is/isn't a zero result).  We can
; use these two facts to copy a single bit (zero extended) to a byte operand.

                test     j, 11000b  ;Test bits 4 and 5.
                setne    Result     ;Result=1 if bits 4 or 5 of J are 1.

                test     k, 10b     ;Test bit #1.
                sete     Result     ;Result=1 if bit #1 = 0.

; The SETcc instructions are particularly useful after a CMP instruction.
; You can set a boolean value according to the result of the comparison.
;
; Result := j <= k

                mov      ax, j
                cmp      ax, k
```

```
                setle   Result      ;Note that "le" is for signed values.

; Result := u1 <= u2

                mov     ax, u1
                cmp     ax, u2
                setbe   Result      ;Note that "be" is for unsigned values.

; One thing nice about the boolean results that the SETcc instructions
; produce is that we can AND, OR, and XOR them and get the same results
; one would expect in a HLL like C, Pascal, or BASIC.
;
; Result := (j < k) and (u1 > u2)

                mov     ax, j
                cmp     ax, k
                setl    bl          ;Use "l" for signed comparisons.

                mov     ax, u1
                cmp     ax, u2
                seta    al          ;Use "a" for unsigned comparisons.

                and     al, bl      ;Logically AND the two boolean results
                mov     Result, al  ; and store the result away.

; Sometimes you can use the shift and rotate instructions to test to see
; if a specific bit is set.  For example, SHR copies bit #0 into the carry
; flag and SHL copies the H.O. bit into the carry flag.  We can easily test
; these bits as follows:
;
; Result := bit #15 of J.

                mov     ax, j
                shl     ax, 1
                setc    Result

; Result := bit #0 of u1:

                mov     ax, u1
                shr     ax, 1
                setc    Result

; If you don't have an 80386 or later processor and cannot use the SETcc
; instructions, you can often simulate them.  Consider the above two
; sequences rewritten for the 8086:

;
; Result := bit #15 of J.

                mov     ax, j
                rol     ax, 1               ;Copy bit #15 to bit #0.
                and     al, 1               ;Strip other bits.
                mov     Result, al

; Result := bit #0 of u1:

                mov     ax, u1
                and     al, 1               ;Strip unnecessary bits.
                mov     Result, al

Quit:           mov     ah, 4ch             ;DOS opcode to quit program.
                int     21h                 ;Call DOS.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends
```

```
zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.6 String Operations

```
; String Instructions

                .386                    ;So we can use extended registers
                option   segment:use16; and addressing modes.

dseg            segment  para public 'data'

String1         byte     "String",0
String2         byte     7 dup (?)

Array1          word     1, 2, 3, 4, 5, 6, 7, 8
Array2          word     8 dup (?)

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax


; The string instructions let you easily copy data from one array to
; another.  If the direction flag is clear, the movsb instruction
; does the equivalent of the following:
;
;       mov es:[di], ds:[si]
;       inc       si
;       inc       di
;
; The following code copies the seven bytes from String1 to String2:

                cld                     ;Required if you want to INC SI/DI

                lea      si, String1
                lea      di, String2

                movsb                   ;String2[0] := String1[0]
                movsb                   ;String2[1] := String1[1]
                movsb                   ;String2[2] := String1[2]
                movsb                   ;String2[3] := String1[3]
                movsb                   ;String2[4] := String1[4]
                movsb                   ;String2[5] := String1[5]
                movsb                   ;String2[6] := String1[6]

; The following code sequence demonstrates how you can use the LODSW and
; STOWS instructions to manipulate array elements during the transfer.
; The following code computes
;
;       Array2[0] := Array1[0]
;       Array2[1] := Array1[0] * Array1[1]
;       Array2[2] := Array1[0] * Array1[1] * Array1[2]
;       etc.
;
; Of course, it would be far more efficient to put the following code
; into a loop, but that will come later.

                lea      si, Array1
                lea      di, Array2
```

```
                        lodsw
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw

                        lodsw
                        imul    ax, dx
                        mov     dx, ax
                        stosw


Quit:                   mov     ah, 4ch             ;DOS opcode to quit program.
                        int     21h                 ;Call DOS.
Main                    endp

cseg                    ends

sseg                    segment para stack 'stack'
stk                     byte    1024 dup ("stack   ")
sseg                    ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               byte    16 dup (?)
zzzzzzseg               ends
                        end     Main
```

6.11.7 Conditional Jumps

```
; Unconditional Jumps

                        .386
                        option  segment:use16

dseg                    segment para public 'data'


; Pointers to statements in the code segment
```

```
IndPtr1         word    IndTarget2
IndPtr2         dword   IndTarget3


dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; JMP instructions transfer control to the
; location specified in the operand field.
; This is typically a label that appears
; in the program.
;
; There are many variants of the JMP
; instruction.  The first is a two-byte
; opcode that transfers control to +/-128
; bytes around the current instruction:

                jmp     CloseLoc
                nop
CloseLoc:


; The next form is a three-byte instruction
; that allows you to jump anywhere within
; the current code segment.  Normally, the
; assembler would pick the shortest version
; of a given JMP instruction, the "near ptr"
; operand on the following instruction
; forces a near (three byte) JMP:


                jmp     near ptr NearLoc
                nop
NearLoc:


; The third form to consider is a five-byte
; instruction that provides a full segmented
; address operand.  This form of the JMP
; instruction lets you transfer control any-
; where in the program, even to another
; segment.  The "far ptr" operand forces
; this form of the JMP instruction:

                jmp     far ptr FarLoc
                nop
FarLoc:


; You can also load the target address of a
; near JMP into a register and jump indirectly
; to the target location.  Note that you can
; use any 80x86 general purpose register to
; hold this address; you are not limited to
; the BX, SI, DI, or BP registers.

                lea     dx, IndTarget
                jmp     dx
                nop
IndTarget:
```

```
; You can even jump indirect through a memory
; variable.  That is, you can jump though a
; pointer variable directly without having to
; first load the pointer variable into a reg-
; ister (Chapter Eight describes why the following
; labels need two colons).

                jmp     IndPtr1
                nop
IndTarget2::


; You can even execute a far jump indirect
; through memory.  Just specify a dword
; variable in the operand field of a JMP
; instruction:

                jmp     IndPtr2
                nop
IndTarget3::



Quit:           mov     ah, 4ch
                int     21h
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.8 CALL and INT Instructions

```
; CALL and INT Instructions

                .386
                option   segment:use16

dseg            segment  para public 'data'

; Some pointers to our subroutines:

SPtr1           word     Subroutine1
SPtr2           dword    Subroutine2

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Subroutine1     proc     near
                ret
Subroutine1     endp

Subroutine2     proc     far
                ret
Subroutine2     endp
```

```
Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Near call:

                call    Subroutine1

; Far call:

                call    Subroutine2

; Near register-indirect call:

                lea     cx, Subroutine1
                call    cx

; Near memory-indirect call:

                call    SPtr1

; Far memory-indirect call:

                call    SPtr2


; INT transfers control to a routine whose
; address appears in the interrupt vector
; table (see the chapter on interrupts for
; details on the interrupt vector table).
; The following call tells the PC's BIOS
; to print theASCII character in AL to the
; display.

                mov     ah, 0eh
                mov     al, 'A'
                int     10h

; INTO generates an INT 4 if the 80x86
; overflow flag is set.  It becomes a
; NOP if the overflow flag is clear.
; You can use this instruction after
; an arithmetic operation to quickly
; test for a fatal overflow.  Note:
; the following sequence does *not*
; generate an overflow.  Do not modify
; it so that it does unless you add an
; INT 4 interrupt service routine to
; the interrupt vector table

                mov     ax, 2
                add     ax, 4
                into


Quit:           mov     ah, 4ch
                int     21h
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
```

```
                end     Main
```

6.11.9 Conditional Jumps I

```
; Conditional JMP Instructions, Part I

                .386
                option  segment:use16
dseg            segment para public 'data'
J               sword   ?
K               sword   ?
L               sword   ?
dseg            ends

cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; 8086 conditional jumps are limited to
; +/- 128 bytes because they are only
; two bytes long (one byte opcode, one
; byte displacement).

                .8086
                ja      lbl
                nop
lbl:

; MASM 6.x will automatically extend out of
; range jumps.  The following are both
; equivalent:

                ja      lbl2
                byte    150 dup (0)
lbl2:
                jna     Temp
                jmp     lbl3
Temp:
                byte    150 dup (0)
lbl3:


; The 80386 and later processors support a
; special form of the conditional jump
; instructions that allow a two-byte displace-
; ment, so MASM 6.x will assemble the code
; to use this form if you've specified an
; 80386 processor.

                .386
                ja      lbl4
                byte    150 dup (0)
lbl4:

; The conditional jump instructions work
; well with the CMP instruction to let you
; execute certain instruction sequences
; only if a condition is true or false.
;
; if (J <= K) then
;      L := L + 1
; else  L := L - 1

                mov     ax, J
```

```
                cmp     ax, K
                jnle    DoElse
                inc     L
                jmp     ifDone

DoElse:         dec     L
ifDone:

; You can also use a conditional jump to
; create a loop in an assembly language
; program:
;
; while (j >= k) do begin
;
;       j := j - 1;
;       k := k + 1;
;       L := j * k;
; end;

WhlLoop:        mov     ax, j
                cmp     ax, k
                jnge    QuitLoop

                dec     j
                inc     k
                mov     ax, j
                imul    ax, k
                mov     L, ax
                jmp     WhlLoop

QuitLoop:

Quit:           mov     ah, 4ch             ;DOS opcode to quit program.
                int     21h                 ;Call DOS.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.11.10 Conditional Jump Instructions II

```
; Conditional JMP Instructions, Part II

                .386
                option   segment:use16
dseg            segment  para public 'data'

Array1          word     1, 2, 3, 4, 5, 6, 7, 8
Array2          word     8 dup (?)

String1         byte     "This string contains lower case characters",0
String2         byte     128 dup (0)

j               sword    5
k               sword    6

Result          byte     ?

dseg            ends
```

```
cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; You can use the LOOP instruction to repeat a sequence of statements
; some specified number of times in an assembly language program.
; Consider the code taken from EX6_5.ASM that used the string
; instructions to produce a running product:
;
; The following code uses a loop instruction to compute:
;
;      Array2[0] := Array1[0]
;      Array2[1] := Array1[0] * Array1[1]
;      Array2[2] := Array1[0] * Array1[1] * Array1[2]
;      etc.

                cld
                lea      si, Array1
                lea      di, Array2
                mov      dx, 1        ;Initialize for 1st time.
                mov      cx, 8        ;Eight elements in the arrays.

LoopHere:lodsw
                imul     ax, dx
                mov      dx, ax
                stosw
                loop     LoopHere


; The LOOPNE instruction is quite useful for controlling loops that
; stop on some condition or when the loop exceeds some number of
; iterations.  For example, suppose string1 contains a sequence of
; characters that end with a byte containing zero.  If you wanted to
; convert those characters to upper case and copy them to string2,
; you could use the following code.  Note how this code ensures that
; it does not copy more than 127 characters from string1 to string2
; since string2 only has enough storage for 127 characters (plus a
; zero terminating byte).

                lea      si, String1
                lea      di, String2
                mov      cx, 127      ;Max 127 chars to string2.

CopyStrLoop:    lodsb                 ;Get char from string1.
                cmp      al, 'a'      ;See if lower case
                jb       NotLower     ;Characters are unsigned.
                cmp      al, 'z'
                ja       NotLower
                and      al, 5Fh      ;Convert lower->upper case.
NotLower:
                stosb
                cmp      al, 0        ;See if zero terminator.
                loopne   CopyStrLoop  ;Quit if al or cx = 0.



; If you do not have an 80386 (or later) CPU and you would like the
; functionality of the SETcc instructions, you can easily achieve
; the same results using code like the following:
;
; Result := J <= K;

                mov      Result, 0    ;Assume false.
                mov      ax, J
                cmp      ax, K
```

```
                jnle    Skip1
                mov     Result, 1    ;Set to 1 if J <= K.
Skip1:


; Result := J = K;

                mov     Result, 0    ;Assume false.
                mov     ax, J
                cmp     ax, K
                jne     Skip2
                mov     Result, 1
Skip2:




Quit:           mov     ah, 4ch              ;DOS opcode to quit program.
                int     21h                  ;Call DOS.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

6.12 Laboratory Exercises

In this set of laboratory exercises you will be writing programs in IBM/L – *Instruction BenchMarking Language.* The IBM/L system lets you time certain instruction sequences to see how long they take to execute.

6.12.1 The IBM/L System

IBM/L lets you time sequences of instructions to see how much time they *really* take to execute. The cycle timings in most 80x86 assembly language books are horribly inaccurate as they assume the absolute best case. IBM/L lets you try out some instruction sequences and see how much time they actually take. This is an invaluable tool to use when optimizing a program. You can try several different instruction sequences that produce the same result and see which sequence executes fastest.

IBM/L uses the system 1/18th second clock and measures most executions in terms of clock ticks. Therefore, it would be totally useless for measuring the speed of a single instruction (since all instructions execute in *much* less than 1/18th second). IBM/L works by repeatedly executing a code sequence thousands (or millions) of times and measuring that amount of time. IBM/L automatically subtracts away the loop overhead time.

IBM/L is a compiler which translates a source language into an assembly language program. Assembling and running the resulting program benchmarks the instructions specified in the IBM/L source code and produces relative timings for different instruction sequences. An IBM/L source program consists of some short assembly language sequences and some control statements which describe how to measure the performance of the assembly sequences. An IBM/L program takes the following form:

```
#data
```

```
        <variable declarations>
#enddata

#unravel <integer constant>
#repetitions <integer constant>
#code ("title")
%init
        <initial instructions whose time does not count>
%eachloop
        <Instructions repeated once on each loop, ignoring time>
%discount
        <instructions done for each sequence, ignoring time>
%do
        <statements to time>
#endcode

<Additional #code..#endcode sections>

#end
```

Note: the %init, %eachloop, and %discount sections are optional.

IBM/L programs begin with an optional data section. The data section begins with a line containing "#DATA" and ends with a line containing "#ENDDATA". All lines between these two lines are copied to an output assembly language program inside the dseg data segment. Typically you would put global variables into the program at this point.

```
Example of a data section:
#DATA
I               word    ?
J               word    ?
K               dword   ?
ch              byte    ?
ch2             byte    ?
#ENDDATA
```

These lines would be copied to a data segment the program IBM/L creates. These names are available to *all* #code..#endcode sequences you place in the program.

Following the data section are one or more code sections. A code section consists of optional #repetition and #unravel statements followed by the actual #code..#endcode sections.

The #repetition statement takes the following form:

```
#repetition integer_constant
```

(The "#" must be in column one). The integer constant is a 32-bit value, so you can specify values in the range zero through two billion. Typical values are generally less than a few hundred thousand, even less on slower machines. The larger this number is, the more accurate the timing will be; however, larger repetition values also cause the program IBM/L generates to run much slower.

This statement instructs IBM/L to generate a loop which repeats the following code segment *integer_constant* times. If you do not specify any repetitions at all, the default is 327,680. Once you set a repetitions value, that value remains in effect for all following code sequences until you explicitly change it again. The #repetition statement must appear outside the #code..#endcode sequence and affects the #code section(s) following the #repetition statement.

If you are interested in the straight-line execution times for some instruction(s), placing those instructions in a tight loop may dramatically affect IBM/L's accuracy. Don't forget, executing a control transfer instruction (necessary for a loop) flushes the pre-fetch queue and has a big effect on execution times. The #unravel statement lets you copy a block of code several times inside the timing loop, thereby reducing the overhead of the conditional jump and other loop control instructions. The #unravel statement takes the following form:

```
#unravel count
```

(The "#" must be in column one). *Count* is a 16-bit integer constant that specifies the number of times IBM/L copies the code inside the repetition loop.

Note that the specified code sequence in the #code section will actually execute (*count* * *integer_constant*) times, since the #unravel statement repeats the code sequence count times inside the loop.

In its most basic form, the #code section looks like the following:

```
#CODE ("Title")
%DO
        <assembly statements>
#ENDCODE
```

The title can be any string you choose. IBM/L will display this title when printing the timing results for this code section. IBM/L will take the specified assembly statements and output them inside a loop (multiple times if the #unravel statement is present). At run time the assembly language source file will time this code and print a time, in clock ticks, for one execution of this sequence.

Example:

```
#unravel 16                 16 copies of code inside the loop
#repetitions 960000         Do this 960,000 times
#code ("MOV AX, 0 Instruction")
%do
                mov     ax, 0
#endcode
```

The above code would generate an assembly language program which executes the mov ax,0 instruction 16 * 960000 times and report the amount of time that it would take.

Most IBM/L programs have multiple code sections. New code sections can immediately follow the previous ones, e.g.,

```
#unravel 16                 16 copies of code inside loop
#repetitions 960000         Do the following code 960000 times
#code ("MOV AX, 0 Instruction")
%do
                mov     ax, 0
#endcode

#code ("XOR AX, AX Instruction")
%do
                xor     ax, ax
 #ENDCODE
```

The above sequence would execute the mov ax, 0 and xor ax, ax instructions 16*960000 times and report the amount of time necessary to execute these instructions. By comparing the results you can determine which instruction sequence is fastest.

Any statement that begins with a semicolon in column one is a comment which IBM/L ignores. It does not write this comment to the assembly language output file.

All IBM/L programs must end with a #end statement. Therefore, the correct form of the program above is

```
#unravel 16
#repetitions 960000
#code ("MOV AX, 0 Instruction")
%do
                mov     ax, 0
#endcode
#code ("XOR AX, AX Instruction")
%do
                xor     ax, ax
#ENDCODE
#END
```

An example of a complete IBM/L program using all of the techniques we've seen so far is

```
#data
                even
i               word    ?
                byte    ?
j               word    ?
#enddata

#unravel 16
#repetitions 32, 30000
#code ("Aligned Word MOV")
%do
                mov     ax, i
#endcode

#code ("Unaligned word MOV")
%do
                mov     ax, j
#ENDCODE
#END
```

There are a couple of optional sections which may appear between the #code and the %do statements. The first of these is %init which begins an initialization section. IBM/L emits initialization sections before the loop, executes this code only once. It does not count their execution time when timing the loop. This lets you set up important values prior to running a test which do not count towards the timing. E.g.,

```
#data
i               dword   ?
#enddata
#repetitions 100000
#unravel 1
#code
%init
                mov     word ptr i, 0
                mov     word ptr i+2, 0
%do
                mov     cx, 200
lbl:            inc     word ptr i
                jnz     NotZero
                inc     word ptr i+2
NotZero:        loop    lbl
#endcode
#end
```

Sometimes you may want to use the #repetitions statement to repeat a section of code several times. However, there may be some statements that you only want to execute once on each loop (that is, without copying the code several times in the loop). The %eachloop section allows this. Note that IBM/L does not count the time consumed by the code executed in the %eachloop section in the final timing.

Example:

```
#data
i               word    ?
j               word    ?
#enddata

#repetitions 40000
#unravel 128
#code
%init -- The following is executed only once

                mov     i, 0
                mov     j, 0

%eachloop -- The following is executed 40000 times, not 128*40000 times

                inc     j

%do -- The following is executed 128 * 40000 times

                inc     i

#endcode
```

```
#end
```

In the above code, IBM/L only counts the time required to increment i. It does not time the instructions in the %init or %eachloop sections.

The code in the %eachloop section only executes once per loop iteration. Even if you use the #unravel statement (the inc i instruction above, for example, executes 128 times per loop iteration because of #unravel). Sometimes you may want some sequence of instructions to execute like those in the %do section, but not count their time. The %discount section allows for this. Here is the full form of an IBM/L source file:

```
#DATA
        <data declarations>
#ENDDATA
#REPETITIONS value1, value2
#UNRAVEL count
#CODE
%INIT
        <Initialization code, executed only once>
%EACHLOOP
        <Loop initialization code, executed once on each pass>
%DISCOUNT
        <Untimed statements, executed once per repetition>
%DO
        <The statements you want to time>
#ENDCODE
<additional code sections>
#END
```

To use this package you need several files. IBML.EXE is the executable program. You run it as follows:

```
c:> IBML filename.IBM
```

This reads an IBML source file (filename.IBM, above) and writes an assembly language program to the standard output. Normally you would use I/O redirection to capture this program as follows:

```
c:> IBML filename.IBM >filename.ASM
```

Once you create the assembly language source file, you can assemble and run it. The resulting EXE file will display the timing results.

To properly run the IBML program, you must have the IBMLINC.A file in the current working directory. This is a skeleton assembly language source file into which IBM/L inserts your assembly source code. Feel free to modify this file as you see fit. Keep in mind, however, that IBM/L expects certain markers in the file (currently ";##") where it will insert the code. Be careful how you deal with these existing markers if you modify the IBMLINC.A file.

The output assembly language source file assumes the presence of the UCR Standard Library for 80x86 Assembly Language Programmers. In particular, it needs the STDLIB include files (stdlib.a) and the library file (stdlib.lib).

In Chapter One of this lab manual you should have learned how to set up the Standard Library files on your hard disk. These must be present in the current directory (or in your INCLUDE/LIB environment paths) or MASM will not be able to properly assemble the output assembly language file. For more information on the UCR Standard Library, see Chapter Seven.

The following are some IBM/L source files to give you a flavor of the language.

```
; IBML Sample program: TESTMUL.IBM.
; This code compares the execution
; time of the MUL instruction vs.
; various shift and add equivalents.

#repetitions 480000
#unravel 1
```

```
; The following check checks to see how
; long it takes to multiply two values
; using the IMUL instruction.

#code ("Multiply by 15 using IMUL")
%do
                .286
                mov     cx, 128
                mov     bx, 15
MulLoop1:       mov     ax, cx
                imul    bx
                loop    MulLoop1

#endcode

; Do the same test using the extended IMUL
; instruction on 80286 and later processors.

#code ("Multiplying by 15 using IMUL")
%do
                mov     cx, 128
MulLoop2:       mov     ax, cx
                imul    ax, 15
                loop    MulLoop2

#endcode

; Now multiply by 15 using a shift by four
; bits and a subtract.

#code ("Multiplying by 15 using shifts and sub")
%init
%do
                mov     cx, 128
MulLoop3:       mov     ax, cx
                mov     bx, ax
                shl     ax, 4
                sub     ax, bx
                loop    MulLoop3

#endcode
#end
```

Output from TESTMUL.IBM:

```
             IBM/L 2.0

Public Domain Instruction Benchmarking Language
 by Randall Hyde, inspired by Roedy Green
All times are measured in ticks, accurate only to ±2.

CPU: 80486

Computing Overhead: Multiply by 15 using IMUL
Testing: Multiply by 15 using IMUL
Multiply by 15 using IMUL :370
Computing Overhead: Multiplying by 15 using IMUL
Testing: Multiplying by 15 using IMUL
Multiplying by 15 using IMUL :370
Computing Overhead: Multiplying by 15 using shifts and sub
Testing: Multiplying by 15 using shifts and sub
Multiplying by 15 using shifts and sub :201
```

```
; IBML Sample program MOVs.
; A comparison of register-register
; moves with register-memory moves

#data
i               word    ?
j               word    ?
k               word    ?
l               word    ?
#enddata
```

```
#repetitions 30720000
#unravel 1

; The following check checks to see how
; long it takes to multiply two values
; using the IMUL instruction.

#code (“Register-Register moves, no Hazards”)
%do
                  mov      bx, ax
                  mov      cx, ax
                  mov      dx, ax
                  mov      si, ax
                  mov      di, ax
                  mov      bp, ax
#endcode

#code (“Register-Register moves, with Hazards”)
%do
                  mov      bx, ax
                  mov      cx, bx
                  mov      dx, cx
                  mov      si, dx
                  mov      di, si
                  mov      bp, di
#endcode

#code (“Memory-Register moves, no Hazards”)
%do
                  mov      ax, i
                  mov      bx, j
                  mov      cx, k
                  mov      dx, l
                  mov      ax, i
                  mov      bx, j
#endcode

#code (“Register-Memory moves, no Hazards”)
%do
                  mov      i, ax
                  mov      j, bx
                  mov      k, cx
                  mov      l, dx
                  mov      i, ax
                  mov      j, bx
#endcode
#end



          IBM/L 2.0

Public Domain Instruction Benchmarking Language
 by Randall Hyde, inspired by Roedy Green
All times are measured in ticks, accurate only to Ò 2.

CPU: 80486

Computing Overhead: Register-Register moves, no Hazards
Testing: Register-Register moves, no Hazards
Register-Register moves, no Hazards :25
Computing Overhead: Register-Register moves, with Hazards
Testing: Register-Register moves, with Hazards
Register-Register moves, with Hazards :51
Computing Overhead: Memory-Register moves, no Hazards
Testing: Memory-Register moves, no Hazards
Memory-Register moves, no Hazards :67
Computing Overhead: Register-Memory moves, no Hazards
Testing: Register-Memory moves, no Hazards
Register-Memory moves, no Hazards :387
```

6.12.2 IBM/L Exercises

The Chapter Six directory contains several sample IBM/L programs (the *.ibm files).. Ex6_1.ibm tests three sequences that compute the absolute value of an integer. Ex6_2.ibm tests three different ways to do a shl left by eight bits. Ex6_3.ibm tests accessing word data at even and odd addresses. Ex6_4.ibm compares the amount of time it takes to load es:bx from a memory location with the time it takes to load es:bx with a constant. Ex6_5.ibm compares the amount of time it takes to swap two registers with and without the XCHG instruction. Ex6_6.ibm compares the multiply instruction against shift & add versions. Ex6_7.ibm compares the speed of register-register move instruction against register-memory move instructions.

Compile each of these IBM/L programs with a DOS command that takes the form:

```
ibml ex6_1.ibm >ex6_1.asm
```

For your lab report: IBM/L writes its output to the standard output device. So use the redirection operator to send this output to a file. Once you've created the file, assemble it with MASM and execute the result. Include the IBM/L program listing and the results in your lab report. **For additional credit:** write your own IBM/L programs to test certain instruction sequences. Include the IBM/L source code in your lab report along with your results.

Warning: to get the most accurate results, you should not run the assembly language programs IBM/L creates under Windows or any other multitasking operating system. For best results, run the output of IBM/L under DOS.

6.13 Programming Projects

1) Write a short "GetLine" routine which reads up to 80 characters from the user and places these characters in successive locations in a buffer in your data segment. Use the INT 16h and INT 10h system BIOS calls described in this chapter to input and output the characters typed by the user. Terminate input upon encountering a carriage return (ASCII code 0Dh) or after the user types the 80th character. Be sure to count the number of characters actually typed by the user for later use. There is a "shell" program specifically designed for this project on the companion CD-ROM (proj6_1.asm).

2) Modify the above routine so that it properly handles the backspace character (ASCII code 08h). Whenever the user presses a backspace key, you should remove the previous keystroke from the input buffer (unless there are no previous characters in the input buffer, in which case you ignore the backspace).

3) You can use the XOR operation to *encrypt* and *decrypt* data. If you XOR all the characters in a message with some value you will effectively *scramble* that message. You can retrieve the original message by XOR'ing the characters in the message with the same value again. Modify the code in Program #2 so that it encrypts each byte in the message with the value 0Fh and displays the encrypted message to the screen. After displaying the message, decrypt it by XOR'ing with 0Fh again and display the decrypted message. Note that you should use the count value computed by the "GetLine" code to determine how many characters to process.

4) Write a "PutString" routine that prints the characters pointed at by the es:di register pair. This routine should print all characters up to (but not including) a zero terminating byte. This routine should preserve all registers it modifies. There is a "shell" program specifically designed for this project on the companion CD-ROM (proj6_4.asm).

5) To output a 16-bit integer value as the corresponding string of decimal digits, you can use the following algorithm:

```
if value = 0 then write('0')
else begin

        DivVal := 10000;
        while (Value mod DivVal) = 0 do begin

                Value := Value mod DivVal;
                DivVal := DivVal div 10;

        end;

        while (DivVal > 1) do begin

                write ( chr( Value div DivVal + 48)); (* 48 = '0' *)
                Value := Value mod DivVal;
                DivVal := DivVal div 10;

        end;
end;
```

Provide a short routine that takes an arbitrary value in ax and outputs it as the corresponding decimal string. Use the int 10h instruction to output the characters to the display. You can use the "shell" program provided on the companion CD-ROM to begin this project (proj6_5.asm).

6) To input a 16-bit integer from the keyboard, you need to use code that uses the following algorithm:

```
Value := 0
repeat

        getchar(ch);
        if (ch >= '0') and (ch <= '9') then begin

          Value := Value * 10 + ord(ch) - ord('0');
        end;
until (ch < '0') or (ch > '9');
```

Use the INT 16h instruction (described in this chapter) to read characters from the keyboard. Use the output routine in program #4 to display the input result. You can use the "shell" file proj6_6.asm to start this project.

6.14 Summary

The 80x86 processor family provides a rich CISC (complex instruction set computer) instruction set. Members of the 80x86 processor family are generally *upward compatible*, meaning successive processors execute all the instructions of the previous chips. Programs written for an 80x86 will generally run on all members of the family, programs using new instructions on the 80286 will run on the 80286 and later processors, but not on the 8086. Likewise, programs that take advantage of the new instructions on the 80386 will run on the 80386 and later processors, but not on the earlier processors. And so on.

The processors described in this chapter include the 8086/8088, the 80286, the 80386, the 80486, and the Pentium (80586). Intel also produced an 80186, but this processor was not used extensively in personal computers[21].

The 80x86 instruction set is most easily broken down into eight categories, see

- "Data Movement Instructions" on page 246.
- "Conversions" on page 252.
- "Arithmetic Instructions" on page 255.
- "Logical, Shift, Rotate and Bit Instructions" on page 269.
- "I/O Instructions" on page 284.
- "String Instructions" on page 284.

21. A few PCs actually used this CPU. Control applications were the biggest user of this CPU, however. The 80186 includes most of the 80286 specific instructions described in this chapter. It does not include the protected mode instructions of the 80286.

- “Program Flow Control Instructions” on page 286.
- “Miscellaneous Instructions” on page 302.

Many instructions affect various flag bits in the 80x86 flags register. Some instructions can test these flags as though they were boolean values. The flags also denote relationships after a comparison such as equality, less than, and greater than. To learn about these flags and how to test them in your programs, consult

- “The Processor Status Register (Flags)” on page 244.
- “The “Set on Condition” Instructions” on page 281.
- “The Conditional Jump Instructions” on page 296.

There are several instructions on the 80x86 that transfer data between registers and memory. These instructions are the ones assembly language programmers use most often. The 80x86 provides many such instructions to help you write fast, efficient, programs. For the details, read

- “Data Movement Instructions” on page 246.
- “The MOV Instruction” on page 246.
- “The XCHG Instruction” on page 247.
- “The LDS, LES, LFS, LGS, and LSS Instructions” on page 248.
- “The LEA Instruction” on page 248.
- “The PUSH and POP Instructions” on page 249.
- “The LAHF and SAHF Instructions” on page 252.

The 80x86 provides several instructions to convert data from one form to another. There are special instructions for sign extension, zero extension, table translation, and big/little endian conversion.

- See “The MOVZX, MOVSX, CBW, CWD, CWDE, and CDQ Instructions” on page 252.
- See “The BSWAP Instruction” on page 254.
- See “The XLAT Instruction” on page 255.

The 80x86 arithmetic instructions provide all the common integer operations: addition, multiplication, subtraction, division, negation, comparisons, increment, decrement, and various instructions to help with BCD arithmetic: AAA, AAD, AAM, AAS, DAA, and DAS. For information on these instructions, see

- “Arithmetic Instructions” on page 255.
- “The Addition Instructions: ADD, ADC, INC, XADD, AAA, and DAA” on page 256.
- “The ADD and ADC Instructions” on page 256.
- “The INC Instruction” on page 258.
- “The XADD Instruction” on page 258.
- “The Subtraction Instructions: SUB, SBB, DEC, AAS, and DAS” on page 259.
- “The CMP Instruction” on page 261.
- “The CMPXCHG, and CMPXCHG8B Instructions” on page 263
- “The NEG Instruction” on page 263.
- “The Multiplication Instructions: MUL, IMUL, and AAM” on page 264.
- “The Division Instructions: DIV, IDIV, and AAD” on page 267.

The 80x86 also provides a rich set of logical, shift, rotate, and bit operations. These instructions manipulate the bits in their operands allowing you to logically AND, OR, XOR, and NOT values, rotate and shift bits within an operand, test and set/clear/invert bits in an operand, and set an operand to zero or one depending on the state of the flags register. For more information, see

- “Logical, Shift, Rotate and Bit Instructions” on page 269.
- “The Logical Instructions: AND, OR, XOR, and NOT” on page 269.
- “The Rotate Instructions: RCL, RCR, ROL, and ROR” on page 276.
- “The Bit Operations” on page 279.
- “The “Set on Condition” Instructions” on page 281.

There are a couple of I/O instruction in the 80x86 instruction set, IN and OUT. These are really special forms of the MOV instruction that operate on the 80x86 I/O address space rather than the memory address space. You typically use these instructions to access hardware registers on peripheral devices. This chapter discusses these instruction at

- "I/O Instructions" on page 284.

The 80x86 family provides a large repertoire of instructions that manipulate strings of data. These instructions include movs, lods, stos, scas, cmps, ins, outs, rep, repz, repe, repnz, and repne. For more information, see

- "String Instructions" on page 284

The transfer of control instructions on the 80x86 let you create loops, subroutines, conditional sequences and do many other tests. To learn about these instructions, read

- "Program Flow Control Instructions" on page 286.
- "Unconditional Jumps" on page 286.
- "The CALL and RET Instructions" on page 289.
- "The INT, INTO, BOUND, and IRET Instructions" on page 292.
- "The Conditional Jump Instructions" on page 296.
- "The JCXZ/JECXZ Instructions" on page 299.
- "The LOOP Instruction" on page 300.
- "The LOOPE/LOOPZ Instruction" on page 300.
- "The LOOPNE/LOOPNZ Instruction" on page 301

This chapter finally discusses various miscellaneous instructions. These instructions directly manipulate flags in the flags register, provide certain processor services, or perform protected mode operations. This Chapter only mentioned the protected mode instructions. Since you do not normally use them in application (non-O/S) programs, you don't really need to know much about them. See

- "Miscellaneous Instructions" on page 302

6.15 Questions

1) Provide an example that shows that it requires n+1 bits to hold the sum of two n-bit binary values.

2) ADC and SBB can be forced to behave exactly like ADD and SUB by inserting some other instruction before ADC and SBB. What instruction must be inserted in front of ADC to make it behave like ADD? In front of SBB to make it behave like SUB?

3) Given that you can manipulate data items on the top of stack using PUSH and POP, explain how you could modify a return address on the top of stack so that a RET instruction will cause the 80x86 to return two bytes beyond the original return address.

4) Provide four different ways to add two to the value in the BX register. No way should require more than two instructions (hint, there are at least six ways of doing this!)

5) Assume that the target addresses for the following conditional jumps are beyond the range of a short branch. Modify each of these instructions so that they perform the proper operation (i.e., can jump the entire distance):

   a) JS Label1 b) JE Label2 c) JZ Label3

   d) JC Label4 e) JBE There f) JG Label5

6) Explain the difference between the carry flag and the overflow flag.

7) When do you typically use the CBW and CWD instructions to sign extend a value?

8) What is the difference between a "MOV reg, immediate" instruction and a "LEA reg, address" instruction?

9) What does the INT nn instruction push onto the stack that the CALL FAR instruction does not?

10) What is the JCXZ instruction typically used for?

11) Explain the operation of the LOOP, LOOPE/LOOPZ, and LOOPNE/LOOPNZ instructions.

12) Which registers (other than the flag register) are affected by the MUL, IMUL, DIV, and IDIV instructions?

13) List three differences between the "DEC AX" and the "SUB AX, 1" instructions.

14) Which of the shift, rotate, and logical instructions do not affect the zero flag?

15) Why does the SAR instruction always clear the overflow flag?

16) On the 80386 the IMUL instruction is almost totally orthogonal (generalized). Almost. Give some examples of forms allowed by the ADD instruction for which there are no comparable IMUL instructions.

17) Why didn't Intel generalize the IDIV instruction as they did the IMUL instruction?

18) What instruction(s) would you need to use to read the eight bit value at I/O address 378h? Please give the specific instructions to do this.

19) Which flag(s) does the 80x86 use to check for *unsigned* arithmetic overflow?

20) Which flag(s) let you check for *signed* overflow?.

21) Which flag(s) does the 80x86 use to test the following *unsigned* conditions? How must the flags be set for the condition to be true?

   a) equal b) not equal c) less than

   d) less than or equal e) greater than f) greater than or equal

22) Repeat the above question for a *signed* comparison.

23) Explain the operation of the 80x86 CALL and RET instructions. Describe step-by-step what each variant of these instructions.

24) The following sequence exchanges the values between the two memory variables I and J:

```
xchg    ax, i
xchg    ax, j
xchg    ax, i
```

On the 80486, the "MOV reg, mem" and "MOV mem, reg" instructions take one cycle (under the right conditions) whereas the "XCHG reg, mem" instruction takes three cycles. Provide a faster sequence for the '486 than the above.

25) On the 80386 the "MOV reg, mem" instruction requires four cycles, the "MOV mem, reg" requires two cycles, and the "XCHG reg, mem" instruction requires five cycles. Provide a faster sequence of the memory exchange problem in question 24 for the 80386.

26) On the 80486, the "MOV acc, mem" and "MOV reg, mem" instructions all take only one cycle to execute (under the right conditions). Assuming all other things equal, why would you want to use the "MOV acc,mem" form rather than the "MOV reg,mem" form to load a value into AL/AX/EAX?

27) Which instructions perform 32 bit loads on the pre-80386 processors?

28) How could you use the PUSH and POP instructions to preserve the AX register between two points in your code?

29) If, immediately upon entering a subroutine, you execute a "pop ax" instruction, what value will you have in the AX register?

30) What is one major use for the SAHF instruction?

31) What is the difference between CWD and CWDE?

32) The BSWAP instruction will convert 32 bit *big endian* values to 32 bit *little endian* values. What instruction can you use to convert 16 bit big endian to 16 bit little endian values?

33) What instruction could you use to convert 32 bit little endian values to 32 bit big endian values?

34) Explain how you could use the XLAT instruction to convert an alphabetic character in the AL register from lower case to upper case (assuming it is lower case) and leave all other values in AL unchanged.

35) What instruction is CMP most similar to?

36) What instruction is TEST most similar to?

37) What does the NEG instruction do?

38) Under what two circumstances will the DIV and IDIV instructions fail?

39) What is the difference between RCL and ROL?

40) Write a short code segment, using the LOOP instruction, that calls the "CallMe" subroutine 25 times.

41) On the 80486 and Pentium CPUs the LOOP instruction is not as fast as the discrete instructions that perform the same operation. Rewrite the code above to produce a faster executing program on the 80486 and Pentium chips.

42) How do you determine the "opposite jump" for a conditional jump. Why is this algorithm preferable?

43) Give an example of the BOUND instruction. Explain what your example will do.

44) What is the difference between the IRET and RET (far) instructions?

45) The BT (Bit Test) instruction copies a specific bit into the carry flag. If the specified bit is one, it sets the carry flag, if the bit is zero, it clears the carry flag. Suppose you want to clear the carry flag if the bit was zero and set it otherwise. What instruction could you execute after BT to accomplish this?

46) You can simulate a far return instruction using a double word variable and two 80x86 instructions. What is the two instruction sequence that will accomplish this?

The UCR Standard Library Chapter Seven

Most programming languages provide several “built-in” functions to reduce the effort needed to write a program. Traditionally, assembly language programmers have not had access to a standard set of commonly used subroutines for their programs; hence, assembly language programmers’ productivity has been quite low because they are constantly “reinventing the wheel” in every program they write. The UCR Standard Library for 80x86 programmers provides such a set of routines. This chapter discusses a small subset of the routines available in the library. After reading this chapter, you should peruse the documentation accompanying the standard library routines.

7.0 Chapter Overview

This chapter provides a basic introduction to the functions available in the UCR Standard Librfary for 80x86 assembly language programmers. This brief introduction covers the following subjects:

- The UCR Standard Library for 80x86 Assembly Language Programmers.
- Memory management routines.
- Input routines.
- Output routines.
- Conversions.
- Predefined constants and macros.

7.1 An Introduction to the UCR Standard Library

The “UCR Standard Library for 80x86 Assembly Language Programmers” is a set of assembly language subroutines patterned after the “C” standard library. Among other things, the standard library includes procedures to handle input, output, conversions, various comparisons and checks, string handling, memory management, character set operators, floating point operations, list handling, serial port I/O, concurrency and coroutines, and pattern matching.

This chapter will not attempt to describe every routine in the library. First of all, the Library is constantly changing so such a description would quickly become outdated. Second, some of the library routines are for advanced programmers only and are beyond the scope of this text. Finally, there are hundreds of routines in the library. Attempting to describe them all here would be a major distraction from the real job at hand– learning assembly language.

Therefore, this chapter will cover the few necessary routines that will get you up and running with the least amount of effort. Note that the full documentation for the library, as well as the source code and several example files are on the companion diskette for this text. A reference guide appears in the appendices of this text. You can also find the latest version of the UCR Standard Library on many on-line services, BBSes, and from many shareware software houses. It is also available via anonymous FTP on the internet.

When using the UCR Standard Library you should always use the SHELL.ASM file provided as the “skeleton” of a new program. This file sets up the necessary segments, provides the proper include directives, and initializes necessary Library routines for you. You should not attempt to create a new program from scratch unless you are very familiar with the internal operation of the Standard Library.

Note that most of the Standard Library routines use macros rather than the call instruction for invocation. You cannot, for example, directly call the putc routine. Instead,

you invoke the putc macro that includes a call to the sl_putc procedure ("SL" stands for "Standard Library").

If you choose not to use the SHELL.ASM file, your program must include several statements to activate the standard library and satisfy certain requirements for the standard library. Please see the documentation accompanying the standard library if you choose to go this route. Until you gain some more experience with assembly language programming, you should always use the SHELL.ASM file as the starting point for your programs.

7.1.1 Memory Management Routines: MEMINIT, MALLOC, and FREE

The Standard Library provides several routines that manage free memory in the *heap*. They give assembly language programmers the ability to dynamically allocate memory during program execution and return this memory to the system when the program no longer needs it. By dynamically allocating and freeing blocks of memory, you can make efficient use of memory on a PC.

The meminit routine initializes the memory manager and you must call it before any routine that uses the memory manager. Since many Standard Library routines use the memory manager, you should call this procedure early in the program. The "SHELL.ASM" file makes this call for you.

The malloc routine allocates storage on the heap and returns a pointer to the block it allocates in the es:di registers. Before calling malloc you need to load the size of the block (in bytes) into the cx register. On return, malloc sets the carry flag if an error occurs (insufficient memory). If the carry is clear, es:di points at a block of bytes the size you've specified:

```
mov       cx, 1024              ;Grab 1024 bytes on the heap
malloc                          ;Call MALLOC
jc        MallocError           ;If memory error.
mov       word ptr PNTR, DI     ;Save away pointer to block.
mov       word ptr PNTR+2, ES
```

When you call malloc, the memory manager promises that the block it gives you is free and clear and it will not reallocate that block until you explicitly free it. To return a block of memory back to the memory manager so you can (possibly) re-use that block of memory in the future, use the free Library routine. free expects you to pass the pointer returned by malloc:

```
les       di, PNTR              ;Get pointer to free
free                            ;Free that block
jc        BadFree
```

As usual for most Standard Library routines, if the free routine has some sort of difficulty it will return the carry flag set to denote an error.

7.1.2 The Standard Input Routines: GETC, GETS, GETSM

While the Standard Library provides several input routines, there are three in particular you will use all the time: getc (get a character), gets (get a string), and getsm (get a malloc'd string).

Getc reads a single character from the keyboard and returns that character in the al register. It returns end of file (EOF) status in the ah register (zero means EOF did not occur, one means EOF did occur). It does not modify any other registers. As usual, the carry flag returns the error status. You do not need to pass getc any values in the registers. Getc does not *echo* the input character to the display screen. You must explicitly print the character if you want it to appear on the output monitor.

The following example program continually loops until the user presses the Enter key:

```
; Note: "CR" is a symbol that appears in the "consts.a"
; header file. It is the value 13 which is the ASCII code
; for the carriage return character

Wait4Enter:     getc
                cmp     al, cr
                jne     Wait4Enter
```

The gets routine reads an entire line of text from the keyboard. It stores each successive character of the input line into a byte array whose base address you pass in the es:di register pair. This array must have room for at least 128 bytes. The gets routine will read each character and place it in the array except for the carriage return character. Gets terminates the input line with a zero byte (which is compatible with the Standard Library string handling routines). Gets echoes each character you type to the display device, it also handles simple line editing functions such as backspace. As usual, gets returns the carry set if an error occurs. The following example reads a line of text from the standard input device and then counts the number of characters typed. This code is tricky, note that it initializes the count and pointer to -1 prior to entering the loop and then immediately increments them by one. This sets the count to zero and adjusts the pointer so that it points at the first character in the string. This simplification produces slightly more efficient code than the straightforward solution would produce:

```
DSEG            segment

MyArray         byte    128 dup (?)

DSEG            ends

CSEG            segment

                .
                .
                .
; Note: LESI is a macro (found in consts.a) that loads
; ES:DI with the address of its operand. It expands to the
; code:
;
;               mov di, seg operand
;               mov es, di
;               mov di, offset operand
;
; You will use the macro quite a bit before many Standard
; Library calls.

                lesi    MyArray             ;Get address of inp buf.
                gets                        ;Read a line of text.
                mov     ah, -1              ;Save count here.
                lea     bx, -1[di]          ;Point just before string.
CountLoop:      inc     ah                  ;Bump count by one.
                inc     bx                  ;Point at next char in str.
                cmp     byte ptr es:[bx], 0
                jne     CoutLoop

; Now AH contains the number of chars in the string.

                .
                .
                .
```

The getsm routine also reads a string from the keyboard and returns a pointer to that string in es:di. The difference between gets and getsm is that you do not have to pass the address of an input buffer in es:di. Getsm automatically allocates storage on the heap with a call to malloc and returns a pointer to the buffer in es:di. Don't forget that you must call meminit at the beginning of your program if you use this routine. The SHELL.ASM skeleton file calls meminit for you. Also, don't forget to call free to return the storage to the heap when you're done with the input line.

```
                getsm                       ;Returns pointer in ES:DI
                .
                .
                .
                free                        ;Return storage to heap.
```

7.1.3 The Standard Output Routines: PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT, and PRINTF

The Standard Library provides a wide array of output routines, far more than you will see here. The following routines are representative of the routines you'll find in the Library.

Putc outputs a single character to the display device. It outputs the character appearing in the al register. It does not affect any registers unless there is an error on output (the carry flag denotes error/no error, as usual). See the Standard Library documentation for more details.

Putcr outputs a "newline" (carriage return/line feed combination) to the standard output. It is completely equivalent to the code:

```
mov     al, cr                  ;CR and LF are constants
putc                            ; appearing in the consts.a
mov     al, lf                  ; header file.
putc
```

The puts (put a string) routine prints the zero terminated string at which es:di points[1]. Note that puts does *not* automatically output a newline after printing the string. You must either put the carriage return/line feed characters at the end of the string or call putcr after calling puts if you want to print a newline after the string. Puts does not affect any registers (unless there is an error). In particular, it does not change the value of the es:di registers. The following code sequence uses this fact:

```
getsm                           ;Read a string
puts                            ;Print it
putcr                           ;Print a new line
free                            ;Free the memory for string.
```

Since the routines above preserve es:di (except, of course, getsm), the call to free deallocates the memory allocated by the call to getsm.

The puth routine prints the value in the al register as exactly two hexadecimal digits, including a leading zero byte if the value is in the range 0..Fh. The following loop reads a sequence of keys from the keyboard and prints their ASCII values until the user presses the Enter key:

```
KeyLoop:        getc
                cmp     al, cr
                je      Done
                puth
                putcr
                jmp     KeyLoop
Done:
```

The puti routine prints the value in ax as a signed 16 bit integer. The following code fragment prints the sum of I and J to the display:

```
mov     ax, I
add     ax, J
puti
putcr
```

Putu is similar to puti except it outputs *unsigned* integer values rather than signed integers.

Routines like puti and putu always output numbers using the minimum number of possible print positions. For example, puti uses three print positions on the string to print the value 123. Sometimes, you may want to force these output routines to print their values using a fixed number of print positions, padding any extra positions with spaces. The putisize and putusize routines provide this capability. These routines expect a numeric value in ax and a field width specification in cx. They will print the number in a field

1. A zero terminated string is a sequence of characters ending with a zero byte. This is the standard character string format the Standard Library uses.

width of *at least* cx positions. If the value in cx is larger than the number of print position the value requires, these routines will right justify the number in a field of cx print positions. If the value in cx is less than the number of print positions the value requires, these routines ignore the value in cx and use however many print positions the number requires.

```
; The following loop prints out the values of a 3x3 matrix in matrix form:
; On entry, bx points at element [0,0] of a row column matrix.

                mov     dx, 3                   ;Repeat for each row.
PrtMatrix:      mov     ax, [bx]                ;Get first element in this
row.
                mov     cx, 7                   ;Use seven print positions.
                putisize                        ;Print this value.
                mov     ax, 2[bx]               ;Get the second element.
                putisize                        ;CX is still seven.
                mov     ax, 4[bx]               ;Get the third element.
                putisize
                putcr                           ;Output a new line.
                add     bx, 6                   ;Move on to next row.
                dec     dx                      ;Repeat for each row.
                jne     PrtMatrix
```

The print routine is one of the most-often called procedures in the library. It prints the zero terminated string that immediately follows the call to print:

```
                print
                byte    "Print this string to the display",cr,lf,0
```

The example above prints the string "Print this string to the display" followed by a new line. Note that print will print whatever characters immediately follow the call to print, up to the first zero byte it encounters. In particular, you can print the newline sequence and any other control characters as shown above. Also note that you are not limited to printing one line of text with the print routine:

```
                print
                byte    "This example of the PRINT routine",cr,lf
                byte    "prints several lines of text.",cr,lf
                byte    "Also note,",cr,lf,"that the source lines "
                byte    "do not have to correspond to the output."
                byte    cr,lf
                byte    0
```

The above displays:

```
This example of the PRINT routine
prints several lines of text.
Also note,
that the source lines do not have to correspond to the output.
```

It is very important that you *not* forget about that zero terminating byte. The print routine begins executing the first 80x86 machine language instruction following that zero terminating byte. If you forget to put the zero terminating byte after your string, the print routine will gladly eat up the instruction bytes following your string (printing them) until it finds a zero byte (zero bytes are common in assembly language programs). This will cause your program to misbehave and is a big source of errors beginning programmers have when they use the print routine. Always keep this in mind.

Printf, like its "C" namesake, provides formatted output capabilities for the Standard Library package. A typical call to printf always takes the following form:

```
                printf
                byte        "format string",0
                dword       operand1, operand2, ..., operandn
```

The format string is comparable to the one provided in the "C" programming language. For most characters, printf simply prints the characters in the format string up to the terminating zero byte. The two exceptions are characters prefixed by a backslash ("\") and characters prefixed by a percent sign ("%"). Like C's printf, the Standard Library's printf

uses the backslash as an escape character and the percent sign as a lead-in to a format string.

Printf uses the escape character ("\") to print special characters in a fashion similar to, but not identical to C's printf. The Standard Library's printf routine supports the following special characters:

- \r Print a carriage return (but no line feed)
- \n Print a new line character (carriage return/line feed).
- \b Print a backspace character.
- \t Print a tab character.
- \l Print a line feed character (but no carriage return).
- \f Print a form feed character.
- \\ Print the backslash character.
- \% Print the percent sign character.
- \0xhh Print ASCII code hh, represented by two hex digits.

C users should note a couple of differences between Standard Library's escape sequences and C's. First, use "\%" to print a percent sign within a format string, not "%%". C doesn't allow the use of "\%" because the C compiler processes "\%" at compile time (leaving a single "%" in the object code) whereas printf processes the format string at run-time. It would see a single "%" and treat it as a format lead-in character. The Standard Library's printf, on the other hand, processes both the "\" and "%" at run-time, therefore it can distinguish "\%".

Strings of the form "\0xhh" must contain exactly two hex digits. The current printf routine isn't robust enough to handle sequences of the form "\0xh" which contain only a single hex digit. Keep this in mind if you find printf chopping off characters after you print a value.

There is absolutely no reason to use any hexadecimal escape character sequence except "\0x00". Printf grabs all characters following the call to printf up to the terminating zero byte (which is why you'd need to use "\0x00" if you want to print the null character, printf will not print such values). The Standard Library's `printf` routine doesn't care how those characters got there. In particular, you are not limited to using a single string after the printf call. The following is perfectly legal:

```
printf
byte "This is a string",13,10
byte "This is on a new line",13,10
byte "Print a backspace at the end of this line:"
byte 8,13,10,0
```

Your code will run a tiny amount faster if you avoid the use of the escape character sequences. More importantly, the escape character sequences take at least two bytes. You can encode most of them as a single byte by simply embedding the ASCII code for that byte directly into the code stream. Don't forget, you cannot embed a zero byte into the code stream. A zero byte terminates the format string. Instead, use the "\0x00" escape sequence.

Format sequences always begin with "%". For each format sequence, you must provide a far pointer to the associated data immediately following the format string, e.g.,

```
printf
byte        "%i %i",0
dword       i,j
```

Format sequences take the general form "%s\cn^f" where:

- "%" is always the "%" character. Use "\%" if you actually want to print a percent sign.
- s is either nothing or a minus sign ("-").
- "\c" is also optional, it may or may not appear in the format item. "c" represents any printable character.
- "n" represents a string of 1 or more decimal digits.
- "^" is just the caret (up-arrow) character.
- "f" represents one of the format characters: i, d, x, h, u, c, s, ld, li, lx, or lu.

The "s", "\c", "n", and "^" items are optional, the "%" and "f" items must be present. Furthermore, the order of these items in the format item is very important. The "\c" entry, for example, cannot precede the "s" entry. Likewise, the "^" character, if present, must follow everything except the "f" character(s).

The format characters i, d, x, h, u, c, s, ld, li, lx, and lu control the output format for the data. The i and d format characters perform identical functions, they tell printf to print the following value as a 16 bit signed decimal integer. The x and h format characters instruct printf to print the specified value as a 16 bit or 8-bit hexadecimal value (respectively). If you specify u, printf prints the value as a 16-bit unsigned decimal integer. Using c tells printf to print the value as a single character. S tells printf that you're supplying the address of a zero-terminated character string, printf prints that string. The ld, li, lx, and lu entries are long (32-bit) versions of d/i, x, and u. The corresponding address points at a 32-bit value that printf will format and print to the standard output.

The following example demonstrates these format items:

```
printf
byte        "I= %i, U= %u, HexC= %h, HexI= %x, C= %c, "
dbyte       "S= %s",13,10
byte        "L= %ld",13,10,0
dword       i,u,c,i,c,s,l
```

The number of far addresses (specified by operands to the "dd" pseudo-opcode) must match the number of "%" format items in the format string. Printf counts the number of "%" format items in the format string and skips over this many far addresses following the format string. If the number of items do not match, the return address for printf will be incorrect and the program will probably hang or otherwise malfunction. Likewise (as for the print routine), the format string must end with a zero byte. The addresses of the items following the format string must point directly at the memory locations where the specified data lies.

When used in the format above, printf always prints the values using the minimum number of print positions for each operand. If you want to specify a minimum field width, you can do so using the "n" format option. A format item of the format "%10d" prints a decimal integer using at least ten print positions. Likewise, "%16s" prints a string using at least 16 print positions. If the value to print requires more than the specified number of print positions, printf will use however many are necessary. If the value to print requires fewer, printf will always print the specified number, padding the value with blanks. Printf will print the value right justified in the print field (regardless of the data's type). If you want to print the value left justified in the output file, use the "-" format character as a prefix to the field width, e.g.,

```
printf
byte        "%-17s",0
dword       string
```

In this example, printf prints the string using a 17 character long field with the string left justified in the output field.

By default, printf blank fills the output field if the value to print requires fewer print positions than specified by the format item. The "\c" format item allows you to change the padding character. For example, to print a value, right justified, using "*" as the padding character you would use the format item "%*10d". To print it left justified you would use the format item "%-*10d". Note that the "-" must precede the "*". This is a limitation of the current version of the software. The operands must appear in this order.

Normally, the address(es) following the printf format string must be far pointers to the actual data to print.

On occasion, especially when allocating storage on the heap (using malloc), you may not know (at assembly time) the address of the object you want to print. You may have only a pointer to the data you want to print. The "^" format option tells printf that the far pointer following the format string is the address of a pointer to the data rather than the address of the data itself. This option lets you access the data indirectly.

Note: unlike C, Standard Library's printf routine does not support floating point output. Putting floating point into printf would increase the size of this routine a tremendous amount. Since most people don't need the floating point output facilities, it doesn't appear here. There is a separate routine, printff, that includes floating point output.

The Standard Library printf routine is a complex beast. However, it is very flexible and extremely useful. You should spend the time to master its major functions. You will be using this routine quite a bit in your assembly language programs.

The standard output package provides many additional routines besides those mentioned here. There simply isn't enough room to go into all of them in this chapter. For more details, please consult the Standard Library documentation.

7.1.4 Formatted Output Routines: Putisize, Putusize, Putlsize, and Putulsize

The puti, putu, and putl routines output the numeric strings using the minimum number of print positions necessary. For example, puti uses three character positions to print the value -12. On occasion, you may need to specify a different field width so you can line up columns of numbers or achieve other formatting tasks. Although you can use printf to accomplish this goal, printf has two major drawbacks – it only prints values in memory (i.e., it cannot print values in registers) and the field width you specify for printf must be a constant[2]. The putisize, putusize, and putlsize routines overcome these limitations.

Like their puti, putu, and putl counterparts, these routines print signed integer, unsigned integer, and 32-bitsigned integer values. They expect the value to print in the ax register (putisize and putusize) or the dx:ax register pair (putlsize). They also expect a minimum field width in the cx register. As with printf, if the value in the cx register is smaller than the number of print positions that the number actually needs to print, putisize, putusize, and putlsize will ignore the value in cx and print the value using the minimum necessary number of print positions.

7.1.5 Output Field Size Routines: Isize, Usize, and Lsize

Once in a while you may want to know the number of print positions a value will require before actually printing that value. For example, you might want to compute the maximum print width of a set of numbers so you can print them in columnar format automatically adjusting the field width for the largest number in the set. The isize, usize, and lsize routines do this for you.

The isize routine expects a signed integer in the ax register. It returns the minimum field width of that value (including a position for the minus sign, if necessary) in the ax register. Usize computes the size of the unsigned integer in ax and returns the minimum field width in the ax register. Lsize computes the minimum width of the signed integer in dx:ax (including a position for the minus sign, if necessary) and returns this width in the ax register.

2. Unless you are willing to resort to self-modifying code.

7.1.6 Conversion Routines: ATOx, and xTOA

The Standard Library provides several routines to convert between string and numeric values. These include atoi, atoh, atou, itoa, htoa, wtoa, and utoa (plus others). The ATOx routines convert an ASCII string in the appropriate format to a numeric value and leave that value in ax or al. The ITOx routines convert the value in al/ax to a string of digits and store this string in the buffer whose address is in es:di[3]. There are several variations on each routine that handle different cases. The following paragraphs describe each routine.

The atoi routine assumes that es:di points at a string containing integer digits (and, perhaps, a leading minus sign). They convert this string to an integer value and return the integer in ax. On return, es:di still points at the beginning of the string. If es:di does not point at a string of digits upon entry or if an overflow occurs, atoi returns the carry flag set. Atoi preserves the value of the es:di register pair. A variant of atoi, atoi2, also converts an ASCII string to an integer except it does *not* preserve the value in the di register. The atoi2 routine is particularly useful if you need to convert a sequence of numbers appearing in the same string. Each call to atoi2 leaves the di register pointing at the first character beyond the string of digits. You can easily skip over any spaces, commas, or other delimiter characters until you reach the next number in the string; then you can call atoi2 to convert that string to a number. You can repeat this process for each number on the line.

Atoh works like the atoi routine, except it expects the string to contain hexadecimal digits (no leading minus sign). On return, ax contains the converted 16 bit value and the carry flag denotes error/no error. Like atoi, the atoh routine preserves the values in the es:di register pair. You can call atoh2 if you want the routine to leave the di register pointing at the first character beyond the end of the string of hexadecimal digits.

Atou converts an ASCII string of decimal digits in the range 0..65535 to an integer value and returns this value in ax. Except that the minus sign is not allowed, this routine behaves just like atoi. There is also an atou2 routine that does not preserve the value of the di register; it leaves di pointing at the first character beyond the string of decimal digits.

Since there is no geti, geth, or getu routines available in the Standard Library, you will have to construct these yourself. The following code demonstrates how to read an integer from the keyboard:

```
print
byte        "Enter an integer value:",0
getsm
atoi        ;Convert string to an integer in AX
free        ;Return storage allocated by getsm
print
byte        "You entered ",0
puti        ;Print value returned by ATOI.
putcr
```

The itoa, utoa, htoa, and wtoa routines are the logical inverse to the ato*x* routines. They convert numeric values to their integer, unsigned, and hexadecimal string representations. There are several variations of these routines depending upon whether you want them to automatically allocate storage for the string or if you want them to preserve the di register.

Itoa converts the 16 bit signed integer in ax to a string and stores the characters of this string starting at location es:di. When you call itoa, you must ensure that es:di points at a character array large enough to hold the resulting string. Itoa requires a maximum of seven bytes for the conversion: five numeric digits, a sign, and a zero terminating byte. Itoa preserves the values in the es:di register pair, so upon return es:di points at the beginning of the string produced by itoa.

Occasionally, you may not want to preserve the value in the di register when calling the itoa routine. For example, if you want to create a single string containing several con-

3. There are also a set of xTOAM routines that automatically allocate storage on the heap for you.

verted values, it would be nice if itoa would leave di pointing at the end of the string rather than at the beginning of the string. The itoa2 routine does this for you; it will leave the di register pointing at the zero terminating byte at the end of the string. Consider the following code segment that will produce a string containing the ASCII representations for three integer variables, Int1, Int2, and Int3:

```
; Assume es:di already points at the starting location to store the converted
; integer values

                mov     ax, Int1
                itoa2                           ;Convert Int1 to a string.

; Okay, output a space between the numbers and bump di so that it points
; at the next available position in the string.

                mov     byte ptr es:[di], ' '
                inc     di

; Convert the second value.

                mov     ax, Int2
                itoa2
                mov     byte ptr es:[di], ' '
                inc     di

; Convert the third value.

                mov     ax, Int3
                itoa2

; At this point, di points at the end of the string containing the
; converted values. Hopefully you still know where the start of the
; string is so you can manipulate it!
```

Another variant of the itoa routine, itoam, does not require you to initialize the es:di register pair. This routine calls malloc to automatically allocate the storage for you. It returns a pointer to the converted string on the heap in the es:di register pair. When you are done with the string, you should call free to return its storage to the heap.

```
; The following code fragment converts the integer in AX to a string and prints
; this string. Of course, you could do this same operation with PUTI, but this
; code does demonstrate how to call itoam.

                itoam                           ;Convert integer to string.
                puts                            ;Print the string.
                free                            ;Return storage to the heap.
```

The utoa, utoa2, and utoam routines work just like itoa, itoa2, and itoam, except they convert the unsigned integer value in ax to a string. Note that utoa and utoa2 require, at most, six bytes since they never output a sign character.

Wtoa, wtoa2, and wtoam convert the 16 bit value in ax to a string of exactly four hexadecimal characters plus a zero terminating byte. Otherwise, they behave exactly like itoa, itoa2, and itoam. Note that these routines output leading zeros so the value is always four digits long.

The htoa, htoa2, and htoam routines are similar to the wtoa, wtoa2, and wtoam routines. However, the hto*x* routines convert the eight bit value in al to a string of two hexadecimal characters plus a zero terminating byte.

The Standard Library provides several other conversion routines as well as the ones mentioned in this section. See the Standard Library documentation in the appendices for more details.

7.1.7 Routines that Test Characters for Set Membership

The UCR Standard Library provides many routines that test the character in the al register to see if it falls within a certain set of characters. These routines all return the status in the zero flag. If the condition is true, they return the zero flag set (so you can test the con-

dition with a je instruction). If the condition is false, they clear the zero flag (test this condition with jne). These routines are

- IsAlNum- Checks to see if al contains an alphanumeric character.
- IsXDigit- Checks al to see if it contains a hexadecimal digit character.
- IsDigit- Checks al to see if it contains a decimal digit character.
- IsAlpha- Checks al to see if it contains an alphabetic character.
- IsLower- Checks al to see if it contains a lower case alpha character.
- IsUpper- Checks al to see if it contains an upper case alpha character.

7.1.8 Character Conversion Routines: ToUpper, ToLower

The ToUpper and ToLower routines check the character in the al register. They will convert the character in al to the appropriate alphabetic case.

If al contains a lower case alphabetic character, ToUpper will convert it to the equivalent upper case character. If al contains any other character, ToUpper will return it unchanged.

If al contains an upper case alphabetic character, ToLower will convert it to the equivalent lower case character. If the value is not an upper case alphabetic character ToLower will return it unchanged.

7.1.9 Random Number Generation: Random, Randomize

The Standard Library Random routine generates a sequence of pseudo-random numbers. It returns a random value in the ax register on each call. You can treat this value as a signed or unsigned value since Random manipulates all 16 bits of the ax register.

You can use the div and idiv instructions to force the output of random to a specific range. Just divide the value random returns by some number *n* and the remainder of this division will be a value in the range 0..*n*-1. For example, to compute a random number in the range 1..10, you could use code like the following:

```
random                  ;Get a random number in range 0..65535.
sub     dx, dx          ;Zero extend to 16 bits.
mov     bx, 10          ;Want value in the range 1..10.
div     bx              ;Remainder goes to dx!
inc     dx              ;Convert 0..9 to 1..10.

; At this point, a random number in the range 1..10 is in the dx register.
```

The random routine always returns the same sequence of values when a program loads from disk and executes. Random uses an internal table of *seed* values that it stores as part of its code. Since these values are fixed, and always load into memory with the program, the algorithm that random uses will always produce the same sequence of values when a program containing it loads from the disk and begins running. This might not seem very “random” but, in fact, this is a nice feature since it is very difficult to test a program that uses truly random values. If a random number generator always produces the same sequence of numbers, any tests you run on that program will be repeatable.

Unfortunately, there are many examples of programs that you may want to write (e.g., games) where having repeatable results is not acceptable. For these applications you can call the randomize routine. Randomize uses the current value of the time of day clock to generate a nearly random starting sequence. So if you need a (nearly) unique sequence of random numbers each time your program begins execution, call the randomize routine once before ever calling the random routine. Note that there is little benefit to calling the randomize routine more than once in your program. Once random establishes a random starting point, further calls to randomize will not improve the quality (randomness) of the numbers it generates.

7.1.10 Constants, Macros, and other Miscellany

When you include the “stdlib.a” header file, you are also defining certain macros (see Chapter Eight for a discussion of macros) and commonly used constants. These include the following:

```
NULL            =       0           ;Some common ASCII codes
BELL            =       07          ;Bell character
bs              =       08          ;Backspace character
tab             =       09          ;Tab character
lf              =       0ah         ;Line feed character
cr              =       0dh         ;Carriage return
```

In addition to the constants above, “stdlib.a” also defines some useful macros including ExitPgm, lesi, and ldxi. These macros contain the following instructions:

```
; ExitPgm- Returns control to MS-DOS

ExitPgm         macro
                mov     ah, 4ch     ;DOS terminate program opcode
                int     21h         ;DOS call.
                endm

; LESI ADRS-
;       Loads ES:DI with the address of the specified operand.

lesi            macro   adrs
                mov     di, seg adrs
                mov     es, di
                mov     di, offset adrs
                endm

; LDXI ADRS-
;       Loads DX:SI with the address of the specified operand.

ldxi            macro   adrs
                mov     dx, seg adrs
                mov     si, offset adrs
                endm
```

The lesi and ldxi macros are especially useful for load addresses into es:di or dx:si before calling various standard library routines (see Chapter Seven for details about macros).

7.1.11 Plus more!

The Standard Library contains many, many, routines that this chapter doesn’t even mention. As you get time, you should read through the documentation for the Standard Library and find out what’s available. The routines mentioned in this chapter are the ones you will use right away. This text will introduce new Standard Library routines as they are needed.

7.2 Sample Programs

The following programs demonstrate some common operations that use the Standard Library.

7.2.1 Stripped SHELL.ASM File

```
; Sample Starting SHELL.ASM file
;
; Randall Hyde
; Version 1.0
; 2/6/96
;
; This file shows what the SHELL.ASM file looks like without
; the superfluous comments that explain where to place objects
; in the source file.  Your programs should likewise begin
; with a stripped version of the SHELL.ASM file.  After all,
; the comments in the original SHELL.ASM file are four *your*
; consumption, not to be read by someone who sees the program
; you wind up writing.

                .xlist
                include    stdlib.a
                includelib  stdlib.lib
                .list

dseg            segment  para public 'data'

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit




Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

7.2.2 Numeric I/O

```
; Pgm7_2.asm - Numeric I/O.
;
; Randall Hyde
; 2/6/96
;
; The standard library routines do not provide simple to use numeric input
; routines.  This code demonstrates how to read decimal and hexadecimal values
; from the user using the Getsm, ATOI, ATOU, ATOH, IsDigit, and IsXDigit
routines.
```

```
                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

dseg            segment  para public 'data'

inputLine       byte     128 dup (0)

SignedInteger   sword    ?
UnsignedInt     word     ?
HexValue        word     ?

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit


; Read a signed integer value from the user.

InputInteger:   print
                byte     "Input a signed integer value: ",0

                lesi     inputLine    ;Point es:di at inputLine buffer
                gets                  ;Read a line of text from the user.

                mov      bx, -1
SkipSpcs1:      inc      bx
                cmp      inputLine[bx], ' '      ;Skip over any spaces.
                je       SkipSpcs1

                cmp      inputLine[bx], '-'      ;See if it's got a minus sign
                jne      NoSign
                inc      bx                      ;Skip if a negative number

NoSign:         dec      bx                      ;Back up one place.
TestDigs:       inc      bx                      ;Move on to next char
                mov      al, inputLine[bx]
                IsDigit                          ;See if it's a decimal digit.
                je       TestDigs                ;Repeat process if it is.

                cmp      inputLine[bx], ' '      ;See if we end with a
                je       GoodDec                 ; reasonable character.
                cmp      inputLine[bx], ','
                je       GoodDec
                cmp      inputLine[bx], 0        ;Input line ends with a zero.
                je       GoodDec
                printf
                byte     "'%s' is an illegal signed integer.  "
                byte     "Please reenter.",cr,lf,0
                dword    inputLine
                jmp      InputInteger

; Okay, all the characters are cool, let's do the conversion here.  Note that
; ES:DI is still pointing at inputLine.

GoodDec:        ATOI                             ;Do the conversion
                mov      SignedInteger, ax       ;Save the value away.


; Read an unsigned integer value from the user.

InputUnsigned:  print
                byte     "Input an unsigned integer value: ",0

                lesi     inputLine    ;Point es:di at inputLine buffer
                gets                  ;Read a line of text from the user.

; Note the sneakiness in the following code.  It starts with an index of -2
; and then increments it by one.  When accessing data in this loop it compares
```

```
; against locatoin inputLine[bx+1] which effectively starts bx at zero.  In the
; "TestUnsigned" loop below, this code increments bx again so that bx then
; contains the index into the string when the action is occuring.

                mov     bx, -2
SkipSpcs2:      inc     bx
                cmp     inputLine[bx+1], ' '    ;Skip over any spaces.
                je      SkipSpcs2

TestUnsigned:   inc     bx                      ;Move on to next char
                mov     al, inputLine[bx]
                IsDigit                         ;See if it's a decimal digit.
                je      TestUnsigned            ;Repeat process if it is.

                cmp     inputLine[bx], ' '      ;See if we end with a
                je      GoodUnSigned            ; reasonable character.
                cmp     inputLine[bx], ','
                je      GoodUnsigned
                cmp     inputLine[bx], 0        ;Input line ends with a zero.
                je      GoodUnsigned
                printf
                byte    "'%s' is an illegal unsigned integer.  "
                byte    "Please reenter.",cr,lf,0
                dword   inputLine
                jmp     InputUnsigned

; Okay, all the characters are cool, let's do the conversion here.  Note that
; ES:DI is still pointing at inputLine.

GoodUnsigned:   ATOU                            ;Do the conversion
                mov     UnsignedInt, ax         ;Save the value away.


; Read a hexadecimal value from the user.

InputHex:       print
                byte    "Input a hexadecimal value: ",0

                lesi    inputLine    ;Point es:di at inputLine buffer
                gets                 ;Read a line of text from the user.

; The following code uses the same sneaky trick as the code above.

                mov     bx, -2
SkipSpcs3:      inc     bx
                cmp     inputLine[bx+1], ' '    ;Skip over any spaces.
                je      SkipSpcs3

TestHex:        inc     bx                      ;Move on to next char
                mov     al, inputLine[bx]
                IsXDigit                        ;See if it's a hex digit.
                je      TestHex                 ;Repeat process if it is.

                cmp     inputLine[bx], ' '      ;See if we end with a
                je      GoodHex                 ; reasonable character.
                cmp     inputLine[bx], ','
                je      GoodHex
                cmp     inputLine[bx], 0        ;Input line ends with a zero.
                je      GoodHex
                printf
                byte    "'%s' is an illegal hexadecimal value.  "
                byte    "Please reenter.",cr,lf,0
                dword   inputLine
                jmp     InputHex

; Okay, all the characters are cool, let's do the conversion here.  Note that
; ES:DI is still pointing at inputLine.

GoodHex:        ATOH                            ;Do the conversion
                mov     HexValue, ax            ;Save the value away.


; Display the results:

                printf
```

```
                        byte    "Values input:",cr,lf
                        byte    "Signed:   %4d",cr,lf
                        byte    "Unsigned: %4d",cr,lf
                        byte    "Hex:      %4x",cr,lf,0
                        dword   SignedInteger, UnsignedInt, HexValue

Quit:                   ExitPgm
Main                    endp

cseg                    ends

sseg                    segment para stack 'stack'
stk                     db      1024 dup ("stack   ")
sseg                    ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               db      16 dup (?)
zzzzzzseg               ends
                        end     Main
```

7.3 Laboratory Exercises

The UCR Standard Library for 80x86 Assembly Language Programmers is available, nearly ready to use, on the companion CD-ROM. In this set of laboratory exercises you will learn how to install the Standard Library on a local hard disk and access the library within your programs.

7.3.1 Obtaining the UCR Standard Library

A recent version of the UCR Standard Library for 80x86 Assembly language programmers appears on the companion CD-ROM. There are, however, periodic updates to the library, so it is quite possible that the version on the CD-ROM is out of date. For most of the projects and examples in this textbook, the version appearing on the CD-ROM is probably sufficient[4]. However, if you want to use the Standard Library to develop your own assembly language software you'll probably want to have the latest version of the library.

The official repository for the UCR Standard library is the ftp.cs.ucr.edu ftp site at the University of California, Riverside. If you have Internet/ftp access, you can download the latest copy of the standard library directly from UCR using an *anonymous ftp account*. To obtain the software over the internet, follow these steps:

- Running your ftp program, connect to ftp.cs.ucr.edu.
- When the system asks for your login name, use *anonymous*.
- When the system asks for your password, use your full login name (e.g., something that looks like *name@machine.domain*).
- At this point, you should be logged onto the system. Switch to the \pub\pc\ibmpcdir using a "cd pub\pc\ibmpcdir" UNIX command.
- The Standard Library files are compressed binary files. Therefore, you must switch ftp to its *binary* (vs. ASCII) mode before downloading the files. On a standard ftp program you would enter a "binary" command to accomplish this. Check the documentation for your ftp program to see how to do this. **The default for download is usually ASCII. If you download the standard library files in ASCII mode, they will probably fail to uncompress properly.**
- In the \pub\pc\ibmpcdir subdirectory you should find several files (generally five but there may be more). Using the appropriate ftp commands (e.g., get or mget), copy these files to your local system.
- Log off the UCR ftp computer and quit your ftp program.

4. Indeed, the only reason to get an update for this text would be to obtain bug fixes.

- If you have been running ftp on a UNIX system, you will need to transfer the files you've downloaded to a PC running DOS or Windows. Consult your instructor or local UNIX system administrator for details.
- That's it! You've now downloaded the latest version of the Standard Library.

If you do not have Internet access, or there is some problem accessing the ftp site at UCR, you can probably locate a copy of the Standard Library at other ftp sites, on other BBSes, or from a shareware vendor. Keep in mind, however, that software you find at other sites may be quite old (indeed, they may have older versions than that appearing on the companion CD-ROM).

For your lab report: If you successfully downloaded the latest version of the library, describe the process you went through. Also, describe the files that you downloaded from the ftp site. If there were any "readme" files you downloaded, read them and describe their content in your lab report.

7.3.2 Unpacking the Standard Library

To reduce disk storage requirements and download time, the UCR Standard Library is compressed. Once you download the files from an ftp site or some other service, you will have to uncompress the files in order to use them. Note: there is a compressed version of the Standard Library on the companion CD-ROM in the event you do not have Internet access and could not download the files in the previous exercises. See the Chapter Seven subdirectory on the companion CD-ROM. Decompressing the Standard Library is nearly an automatic process. Just follow these steps:

- Create a directory on your local hard disk (usually C:) named "STDLIB".[5] Switch to this subdirectory using the command "CD C:\STDLIB".
- Copy the files you downloaded (or the files off the companion CD-ROM in the STDLIB\DIST subdirectory) into the STDLIB subdirectory you've just created.
- Execute the DOS command "PATH=C:\STDLIB".
- Execute the "UNPACK.BAT" batch file by typing "UNPACK" at the DOS command line prompt.
- Sit back and watch the show. Everything else is automatic.
- You should reboot after unpacking the standard library or reset the path to its original value.

If you did not set the path to include the STDLIB directory, the UNPACK.BAT file will report several errors and it will not properly unpack the files. It *will* delete the compressed files from the disk. Therefore, make sure you save a copy of the files you downloaded on a floppy disk or in a different directory when unpacking the Standard Library. Doing so will save you from having to download the STDLIB files again if something goes wrong during the decompression phase.

For your lab report: Describe the directory structure that unpacking the standard library produces.

7.3.3 Using the Standard Library

When you unpack the Standard Library files, the UNPACK.BAT program leaves a (full) copy of the SHELL.ASM file sitting in the STDLIB subdirectory. This should be a familiar file since you've been using SHELL.ASM as a skeletal assembly language program in past projects. This particular version of SHELL.ASM is a "full" version since it

5. If you are doing this on computer systems in your school's laboratories, they may ask you to use a different subdirectory since the Standard Library may already be installed on the machines.

contains several comments that explain where user-written code and variables should go in the file. As a general rule, it is very bad programming style to leave these comments in your SHELL.ASM file. Once you've read these comments and figured out the layout of the SHELL.ASM file, you should delete those comments from any program you write based on the SHELL.ASM file.

For your lab report: include a modified version of the SHELL.ASM file with the superfluous comments removed.

At the beginning of the SHELL.ASM file, you will find the following two statements:

```
include     stdlib.a
includelib  stdlib.lib
```

The first statement tells MASM to read the definitions for the standard library routines from the STDLIB.A *include file* (see Chapter Eight for a description of include files). The second statement tells MASM to pass the name of the STDLIB.LIB object code file on to the linker so it can link your program with the code in the Standard Library. The exact nature of these two statements is unimportant at this time; however, to use the Standard Library routines, MASM needs to be able to *find* these two files at assembly and link time. By default, MASM assumes that these two files are in the current subdirectory whenever you assemble a program based on SHELL.ASM. Since this is not the case, you will have to execute two special DOS commands to tell MASM where it can find these files. The two commands are

```
set include=c:\stdlib\include
set lib=c:\stdlib\lib
```

If you do not execute these commands at least once prior to using MASM with SHELL.ASM for the first time, MASM will report an error (that it cannot find the STDLIB.A file) and abort the assembly.

For your lab report: Execute the DOS commands "SET INCLUDE=C:\" and "SET LIB=C:\"[6] and then attempt to assemble SHELL.ASM using the DOS command:

```
ml shell.asm
```

Report the error in your lab report. Now execute

```
SET INCLUDE=C:\STDLIB\INCLUDE
```

Assemble SHELL.ASM again and report any errors. Finally, execute LIB set command and assemble your program (hopefully) without error.

If you want to avoid having to execute the SET commands every time you sit down to program in assembly language, you can always add these set commands to your autoexec.bat file. If you do this, the system will automatically execute these commands whenever you turn it on.

Other programs (like MASM and Microsoft C++) may also be using SET LIB and SET INCLUDE commands. If there are already SET INCLUDE or SET LIB commands in your autoexec.bat file, you should append the Standard Library entrys to the end of the existing command like the following:

```
set include=c:\MASM611\include;c:\STDLIB\INCLUDE
set lib=c:\msvc\lib;C:\STDLIB\LIB
```

7.3.4 The Standard Library Documentation Files

There are several hundred routines in the UCR Standard Library; far more than this chapter can reasonably document. The "official" source of documentation for the UCR Standard Library is a set of text files appearing in the C:\STDLIB\DOC directory. These files are text files (that you can read with any text editor) that describe the use of each of

6. These command deactivate any current LIB or INCLUDE strings in the environment variables.

the Standard Library routines. If you have any questions about a subroutine or you want to find out what routines are available, you should read the files in this subdirectory.

The documentation consists of several text files organized by routine classification. For example, one file describes output routines, another describes input routines, and yet another describes the string routines. The SHORTREF.TXT file provides a quick synopsis of the entire library. This is a good starting point for information about the routines in the library.

For your lab report: include the names of the text files appearing in the documentation directory. Provide the names of several routines that are docuemented within each file.

7.4 Programming Projects

1) Write any program of your choice that uses at least fifteen different UCR Standard Library routines. Consult the appendix in your textbook and the STDLIB\DOC directory for details on the various StdLib routines. At least five of the routines you choose should *not* appear in this chapter. Learn those routines yourself by studying the UCR StdLib documentation.

2) Write a program that demonstrates the use of each of the format options in the PRINTF StdLib routine.

3) Write a program that reads 16 signed integers from a user and stores these values into a 4x4 matrix. The program should then print the 4x4 matrix in matrix form (i.e., four rows of four numbers with each column nicely aligned).

4) Modify the program in problem (3) above so that figures out which number requires the largest number of print positions and then it outputs the matrix using this value plus one as the field width for all the numbers in the matrix. For example, if the largest number in the matrix is 1234, then the program would print the numbers in the matrix using a minimum field width of five.

7.5 Summary

This chapter introduced several assembler directives and pseudo-opcodes supported by MASM. It also briefly discussed some routines in the UCR Standard Library for 80x86 Assembly Language Programmers. This chapter, by no means, is a complete description of what MASM or the Standard Library has to offer. It does provide enough information to get you going.

To help you write assembly language programs with a minimum of fuss, this text makes extensive use of various routines from the UCR Standard Library for 80x86 Assembly Language Programmers. Although this chapter could not possibly begin to cover all the Standard Library routines, it does discuss many of the routines that you'll use right away. This text will discuss other routines as necessary.

- See "An Introduction to the UCR Standard Library" on page 333.
- See "Memory Management Routines: MEMINIT, MALLOC, and FREE" on page 334.
- See "The Standard Input Routines: GETC, GETS, GETSM" on page 334.
- See "The Standard Output Routines: PUTC, PUTCR, PUTS, PUTH, PUTI, PRINT, and PRINTF" on page 336.
- See "Conversion Routines: ATOx, and xTOA" on page 341.
- "Formatted Output Routines: Putisize, Putusize, Putlsize, and Putulsize" on page 340
- "Output Field Size Routines: Isize, Usize, and Lsize" on page 340
- "Routines that Test Characters for Set Membership" on page 342

- “Character Conversion Routines: ToUpper, ToLower” on page 343
- “Random Number Generation: Random, Randomize” on page 343
- “Constants, Macros, and other Miscellany” on page 344
- See“Plus more!” on page 344.

7.6 Questions

1. What file should you use to begin your programs when writing code that uses the UCR Standard Library?
2. What routine allocates storage on the heap?
3. What routine would you use to print a single character?
4. What routines allow you to print a literal string of characters to the display?
5. The Standard Library does not provide a routine to read an integer from the user. Describe how to use the GETS and ATOI routines to accomplish this task.
6. What is the difference between the GETS and GETSM routines?
7. What is the difference between the ATOI and ATOI2 routines?
8. What does the ITOA routine do? Describe input and output values.

MASM: Directives & Pseudo-Opcodes Chapter Eight

Statements like mov ax,0 and add ax,bx are meaningless to the microprocessor. As arcane as these statements appear, they are still human readable forms of 80x86 instructions. The 80x86 responds to commands like B80000 and 03C3. An assembler is a program that converts strings like mov ax,0 to 80x86 machine code like "B80000". An assembly language program consists of statements like mov ax,0. The assembler converts an assembly language source file to machine code – the binary equivalent of the assembly language program. In this respect, the assembler program is much like a compiler, it reads an ASCII source file from the disk and produces a machine language program as output. The major difference between a compiler for a high level language (HLL) like Pascal and an assembler is that the compiler usually emits several machine instructions for each Pascal statement. The assembler generally emits a single machine instruction for each assembly language statement.

Attempting to write programs in machine language (i.e., in binary) is not particularly bright. This process is very tedious, prone to mistakes, and offers almost no advantages over programming in assembly language. The only major disadvantage to assembly language over pure machine code is that you must first assemble and link a program before you can execute it. However, attempting to assemble the code by hand would take far longer than the small amount of time that the assembler takes to perform the conversion for you.

There is another disadvantage to learning assembly language. An assembler like Microsoft's Macro Assembler (MASM) provides a large number of features for assembly language programmers. Although learning about these features takes a fair amount of time, they are so useful that it is well worth the effort.

8.0 Chapter Overview

Like Chapter Six, much of the information in this chapter is reference material. Like any reference section, some knowledge is essential, other material is handy, but optional, and some material you may never use while writing programs. The following list outlines the information in this text. A "•" symbol marks the essential material. The "❑" symbol marks the optional and lesser used subjects.

- • Assembly language statement source format
- ❑ The location counter
- • Symbols and identifiers
- • Constants
- • Procedure declarations
- ❑ Segments in an assembly language program
- • Variables
- • Symbol types
- • Address expressions (later subsections contain advanced material)
- ❑ Conditional assembly
- ❑ Macros
- ❑ Listing directives
- ❑ Separate assembly

8.1 Assembly Language Statements

Assembly language statements in a source file use the following format:

```
{Label}          {Mnemonic    {Operand}}        {;Comment}
```

Each entity above is a field. The four fields above are the *label field*, the *mnemonic field*, the *operand field*, and the *comment field*.

The label field is (usually) an optional field containing a symbolic label for the current statement. Labels are used in assembly language, just as in HLLs, to mark lines as the targets of GOTOs (jumps). You can also specify variable names, procedure names, and other entities using symbolic labels. Most of the time the label field is optional, meaning a label need be present only if you want a label on that particular line. Some mnemonics, however, require a label, others do not allow one. In general, you should always begin your labels in column one (this makes your programs easier to read).

A mnemonic is an instruction name (e.g., mov, add, etc.). The word mnemonic means memory aid. mov is much easier to remember than the binary equivalent of the mov instruction! The braces denote that this item is optional. Note, however, that you cannot have an operand without a mnemonic.

The mnemonic field contains an assembler instruction. Instructions are divided into three classes: 80x86 machine instructions, assembler directives, and pseudo opcodes. 80x86 instructions, of course, are assembler mnemonics that correspond to the actual 80x86 instructions introduced in Chapter Six.

Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equ, assume, and end. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, nothing else.

A pseudo-opcode is a message to the assembler, just like an assembler directive, however a pseudo-opcode will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and tbyte. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

The operand field contains the operands, or parameters, for the instruction specified in the mnemonic field. Operands never appear on lines by themselves. The type and number of operands (zero, one, two, or more) depend entirely on the specific instruction.

The comment field allows you to annotate each line of source code in your program. Note that the comment field always begins with a semicolon. When the assembler is processing a line of text, it completely ignores everything on the source line following a semicolon[1].

Each assembly language statement appears on its own line in the source file. You cannot have multiple assembly language statements on a single line. On the other hand, since all the fields in an assembly language statement are optional, blank lines are fine. You can use blank lines anywhere in your source file. Blank lines are useful for spacing out certain sections of code, making them easier to read.

The Microsoft Macro Assembler is a free form assembler. The various fields of an assembly language statement may appear in any column (as long as they appear in the proper order). Any number of spaces or tabs can separate the various fields in the statement. To the assembler, the following two code sequences are identical:

```
                mov     ax, 0
                mov     bx, ax
                add     ax, dx
                mov     cx, ax
```

```
mov             ax,                 0
        mov bx,                     ax
    add             ax, dx
                        mov         cx, ax
```

1. Unless, of course, the semicolon appears inside a string constant.

The first code sequence is much easier to read than the second (if you don't think so, perhaps you should go see a doctor!). With respect to readability, the judicial use of spacing within your program can make all the difference in the world.

Placing the labels in column one, the mnemonics in column 17 (two tabstops), the operand field in column 25 (the third tabstop), and the comments out around column 41 or 49 (five or six tabstops) produces the best looking listings. Assembly language programs are hard enough to read as it is. Formatting your listings to help make them easier to read will make them much easier to maintain.

You may have a comment on the line by itself. In such a case, place the semicolon in column one and use the entire line for the comment, examples:

```
; The following section of code positions the cursor to the upper
; left hand position on the screen:

                mov     X, 0
                mov     Y, 0

; Now clear from the current cursor position to the end of the
; screen to clear the video display:

;               etc.
```

8.2 The Location Counter

Recall that all addresses in the 80x86's memory space consist of a segment address and an offset within that segment. The assembler, in the process of converting your source file into object code, needs to keep track of offsets within the current segment. The *location counter* is an assembler variable that handles this.

Whenever you create a segment in your assembly language source file (see segments later in this chapter), the assembler associates the current location counter value with it. The location counter contains the current offset into the segment. Initially (when the assembler first encounters a segment) the location counter is set to zero. When encountering instructions or pseudo-opcodes, MASM increments the location counter for each byte written to the object code file. For example, MASM increments the location counter by two after encountering mov ax, bx since this instruction is two bytes long.

The value of the location counter varies throughout the assembly process. It changes for each line of code in your program that emits object code. We will use the term location counter to mean the value of the location counter at a particular statement before generating any code. Consider the following assembly language statements:

```
0 :             or      ah, 9
3 :             and     ah, 0c9h
6 :             xor     ah, 40h
9 :             pop     cx
A :             mov     al, cl
C :             pop     bp
D :             pop     cx
E :             pop     dx
F :             pop     ds
10:             ret
```

The or, and, and xor instructions are all three bytes long; the mov instruction is two bytes long; the remaining instructions are all one byte long. If these instructions appear at the beginning of a segment, the location counter would be the same as the numbers that appear immediately to the left of each instruction above. For example, the or instruction above begins at offset zero. Since the or instruction is three bytes long, the next instruction (and) follows at offset three. Likewise, and is three bytes long, so xor follows at offset six, etc..

8.3 Symbols

Consider the jmp instruction for a moment. This instruction takes the form:

```
jmp   target
```

Target is the destination address. Imagine how painful it would be if you had to actually specify the target memory address as a numeric value. If you've ever programmed in BASIC (where line numbers are the same thing as statement labels) you've experienced about 10% of the trouble you would have in assembly language if you had to specify the target of a jmp by an address.

To illustrate, suppose you wanted to jump to some group of instructions you've yet to write. What is the address of the target instruction? How can you tell until you've written every instruction before the target instruction? What happens if you change the program (remember, inserting and deleting instructions will cause the location counter values for all the following instructions within that segment to change). Fortunately, all these problems are of concern only to machine language programmers. Assembly language programmers can deal with addresses in a much more reasonable fashion – by using symbolic addresses.

A *symbol*, *identifier*, or *label* , is a name associated with some particular value. This value can be an offset within a segment, a constant, a string, a segment address, an offset within a record, or even an operand for an instruction. In any case, a label provides us with the ability to represent some otherwise incomprehensible value with a familiar, mnemonic, name.

A symbolic name consists of a sequence of letters, digits, and special characters, with the following restrictions:

- A symbol cannot begin with a numeric digit.
- A name can have any combination of upper and lower case alphabetic characters. The assembler treats upper and lower case equivalently.
- A symbol may contain any number of characters, however only the first 31 are used. The assembler ignores all characters beyond the 31st.
- The _, $, ?, and @ symbols may appear anywhere within a symbol. However, $ and ? are special symbols; you cannot create a symbol made up solely of these two characters.
- A symbol cannot match any name that is a reserved symbol. The following symbols are reserved:

```
%out            .186            .286            .286P
.287            .386            .386P           .387
.486            .486P           .8086           .8087
.ALPHA          .BREAK          .CODE           .CONST
.CREF           .DATA           .DATA?          .DOSSEG
.ELSE           .ELSEIF         .ENDIF          .ENDW
.ERR            .ERR1           .ERR2           .ERRB
.ERRDEF         .ERRDIF         .ERRDIFI        .ERRE
.ERRIDN         .ERRIDNI        .ERRNB          .ERRNDEF
.ERRNZ          .EXIT           .FARDATA        .FARDATA?
.IF             .LALL           .LFCOND         .LIST
.LISTALL        .LISTIF         .LISTMACRO      .LISTMACROALL
.MODEL          .MSFLOAT        .NO87           .NOCREF
.NOLIST         .NOLISTIF       .NOLISTMACRO    .RADIX
.REPEAT         .UNTIL          .SALL           .SEQ
.SFCOND         .STACK          .STARTUP        .TFCOND
.UNTIL          .UNTILCXZ       .WHILE          .XALL
.XCREF          .XLIST          ALIGN           ASSUME
BYTE            CATSTR          COMM            COMMENT
DB              DD              DF              DOSSEG
DQ              DT              DW              DWORD
ECHO            ELSE            ELSEIF          ELSEIF1
ELSEIF2         ELSEIFB         ELSEIFDEF       ELSEIFDEF
ELSEIFE         ELSEIFIDN       ELSEIFNB        ELSEIFNDEF
```

```
END             ENDIF           ENDM            ENDP
ENDS            EQU             EVEN            EXITM
EXTERN          EXTRN           EXTERNDEF       FOR
FORC            FWORD           GOTO            GROUP
IF              IF1             IF2             IFB
IFDEF           IFDIF           IFDIFI          IFE
IFIDN           IFIDNI          IFNB            IFNDEF
INCLUDE         INCLUDELIB      INSTR           INVOKE
IRP             IRPC            LABEL           LOCAL
MACRO           NAME            OPTION          ORG
PAGE            POPCONTEXT      PROC            PROTO
PUBLIC          PURGE           PUSHCONTEXT     QWORD
REAL4           REAL8           REAL10          RECORD
REPEAT          REPT            SBYTE           SDWORD
SEGMENT         SIZESTR         STRUC           STRUCT
SUBSTR          SUBTITLE        SUBTTL          SWORD
TBYTE           TEXTEQU         TITLE           TYPEDEF
UNION           WHILE           WORD
```

In addition, all valid 80x86 instruction names and register names are reserved as well. Note that this list applies to Microsoft's Macro Assembler version 6.0. Earlier versions of the assembler have fewer reserved words. Later versions may have more.

Some examples of valid symbols include:

```
L1              Bletch      RightHere
Right_Here      Item1       __Special
$1234           @Home       $_@1
Dollar$         WhereAmI?   @1234
```

$1234 and @1234 are perfectly valid, strange though they may seem.

Some examples of illegal symbols include:

```
1TooMany        - Begins with a digit.
Hello.There     - Contains a period in the middle of the symbol.
$               - Cannot have $ or ? by itself.
LABEL           - Assembler reserved word.
Right Here      - Symbols cannot contain spaces.
Hi,There        - or other special symbols besides _, ?, $, and @.
```

Symbols, as mentioned previously, can be assigned numeric values (such as location counter values), strings, or even whole operands. To keep things straightened out, the assembler assigns a type to each symbol. Examples of types include near, far, byte, word, double word, quad word, text, and strings. How you declare labels of a certain type is the subject of much of the rest of this chapter. For now, simply note that the assembler always assigns some type to a label and will tend to complain if you try to use a label at some point where it does not allow that type of label.

8.4 Literal Constants

The Microsoft Macro Assembler (MASM) is capable of processing five different types of constants: integers, packed binary coded decimal integers, real numbers, strings, and text. In this chapter we'll consider integers, reals, strings, and text only. For more information about packed BCD integers please consult the Microsoft Macro Assembler Programmer's Guide.

A *literal constant* is one whose value is implicit from the characters that make up the constant. Examples of literal constants include:

- 123
- 3.14159
- “Literal String Constant”
- 0FABCh
- ‘A’
- <Text Constant>

Except for the last example above, most of these literal constants should be reasonably familiar to anyone who has written a program in a high level language like Pascal or C++. Text constants are special forms of strings that allow textual substitution during assembly.

A literal constant's representation corresponds to what we would normally expect for its "real world value." Literal constants are also known as *non symbolic constants* since they use the value's actual representation, rather than some symbolic name, within your program. MASM also lets you define symbolic, or *manifest*, constants in a program, but more on that later.

8.4.1 Integer Constants

An integer constant is a numeric value that can be specified in binary, decimal, or hexadecimal[2]. The choice of the base (or radix) is up to you. The following table shows the legal digits for each radix:

Table 35: Digits Used With Each Radix

Name	Base	Valid Digits
Binary	2	0 1
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

To differentiate between numbers in the various bases, you use a suffix character. If you terminate a number with a "b" or "B", then MASM assumes that it is a binary number. If it contains any digits other than zero or one the assembler will generate an error. If the suffix is "t", "T", "d" or "D", then the assembler assumes that the number is a decimal (base 10) value. A suffix of "h" or "H" will select the hexadecimal radix.

All integer constants must begin with a decimal digit, including hexadecimal constants. To represent the value "FDED" you must specify 0FDEDh. The leading decimal digit is required by the assembler so that it can differentiate between symbols and numeric constants; remember, "FDEDh" is a perfectly valid symbol to the Microsoft Macro Assembler.

Examples:

```
0F000h      12345d              0110010100b
1234h       100h                08h
```

If you do not specify a suffix after your numeric constants, the assembler uses the current default radix. Initially, the default radix is decimal. Therefore, you can usually specify decimal values without the trailing "D" character. The radix assembler directive can be used to change the default radix to some other base. The .radix instruction takes the following form:

```
.radix      base                ;Optional comment
```

Base is a decimal value between 2 and 16.

The .radix statement takes effect as soon as MASM encounters it in the source file. All the statements before the .radix statement will use the previous default base for numeric constants. By sprinkling multiple .radix instructions throughout your source file, you can switch the default base amongst several values depending upon what's most convenient at each point in your program.

Generally, decimal is fine as the default base so the .radix instruction doesn't get used much. However, faced with entering a gigantic table of hexadecimal values, you can save

2. Actually, you can also specify the octal (base 8) radix. We will not use octal in this text.

a lot of typing by temporarily switching to base 16 before the table and switching back to decimal after the table. Note: if the default radix is hexadecimal, you should use the “T” suffix to denote decimal values since MASM will confuse the “D” suffix with a hexadecimal digit.

8.4.2 String Constants

A string constant is a sequence of characters surrounded by apostrophes or quotation marks.

Examples:

```
"This is a string"
 'So is this'
```

You may freely place apostrophes inside string constants enclosed by quotation marks and vice versa. If you want to place an apostrophe inside a string delimited by apostrophes, you must place a pair of apostrophes next to each other in the string, e.g.,

```
'Doesn''t this look weird?'
```

Quotation marks appearing within a string delimited by quotes must also be doubled up, e.g.,

```
"Microsoft claims ""Our software is very fast.""  Do you believe them?"
```

Although you can double up apostrophes or quotes as shown in the examples above, the easiest way to include these characters in a string is to use the *other* character as the string delimiter:

```
                “Doesn’t this look weird?”
‘Microsoft claims “Our software is very fast.” Do you believe them?’
```

The only time it would be absolutely necessary to double up quotes or apostrophes in a string is if that string contained *both* symbols. This rarely happens in real programs.

Like the C and C++ programming languages, there is a subtle difference between a character value and a string value. A single character (that is, a string of length one) may appear anywhere MASM allows an integer constant or a string. If you specify a character constant where MASM expects an integer constant, MASM uses the ASCII code of that character as the integer value. Strings (whose length is greater than one) are allowed only within certain contexts.

8.4.3 Real Constants

Within certain contexts, you can use floating point constants. MASM allows you to express floating point constants in one of two forms: decimal notation or scientific notation. These forms are quite similar to the format for real numbers that Pascal, C, and other HLLs use.

The decimal form is just a sequence of digits containing a decimal point in some position of the number:

```
1.0       3.14159     625.25                -128.0        0.5
```

Scientific notation is also identical to the form used by various HLLs:

```
1e5       1.567e-2    -6.02e-10             5.34e+12
```

The exact range of precision of the numbers depend on your particular floating point package. However, MASM generally emits binary data for the above constants that is compatible with the 80x87 numeric coprocessors. This form corresponds to the numeric format specified by the IEEE standard for floating point values. In particular, the constant 1.0 is not the binary equivalent of the integer one.

8.4.4 Text Constants

Text constants are not the same thing as string constants. A textual constant substitutes verbatim during the assembly process. For example, the characters 5[bx] could be a textual constant associated with the symbol VAR1. During assembly, an instruction of the form mov ax, VAR1 would be converted to the instruction mov ax, 5[bx].

Textual equates are quite useful in MASM because MASM often insists on long strings of text for some simple assembly language operands. Using text equates allows you to simplify such operands by substituting some string of text for a single identifier in a statement.

A text constant consists of a sequence of characters surrounded by the "<" and ">" symbols. For example the text constant 5[bx] would normally be written as <5[bx]>. When the text substitution occurs, MASM strips the delimiting "<" and ">" characters.

8.5 Declaring Manifest Constants Using Equates

A manifest constant is a symbol name that represents some fixed quantity during the assembly process. That is, it is a symbolic name that represents some value. *Equates* are the mechanism MASM uses to declare symbolic constants. Equates take three basic forms:

```
symbol          equ      expression
symbol          =        expression
symbol          textequ  expression
```

The expression operand is typically a numeric expression or a text string. The symbol is given the value and type of the expression. The equ and "=" directives have been with MASM since the beginning. Microsoft added the textequ directive starting with MASM 6.0.

The purpose of the "=" directive is to define symbols that have an integer (or single character) quantity associated with them. This directive does not allow real, string, or text operands. This is the primary directive you should use to create numeric symbolic constants in your programs. Some examples:

```
NumElements     =        16
                  .
                  .
                  .
Array           byte     NumElements dup (?)
                  .
                  .
                  .
                mov      cx, NumElements
                mov      bx, 0
ClrLoop:        mov      Array[bx], 0
                inc      bx
                loop     ClrLoop
```

The textequ directive defines a text substitution symbol. The expression in the operand field must be a text constant delimited with the "<" and ">" symbols. Whenever MASM encounters the symbol within a statement, it substitutes the text in the operand field for the symbol. Programmers typically use this equate to save typing or to make some code more readable:

```
Count           textequ  <6[bp]>
DataPtr         textequ  <8[bp]>
                  .
                  .
                  .
                les      bx, DataPtr  ;Same as les bx, 8[bp]
                mov      cx, Count    ;Same as mov cx, 6[bp]
                mov      al, 0
ClrLp:          mov      es:[bx], al
                inc      bx
                loop     ClrLp
```

Note that it is perfectly legal to equate a symbol to a blank operand using an equate like the following:

```
BlankEqu        textequ  <>
```

The purpose of such an equate will become clear in the sections on conditional assembly and macros.

The equ directive provides almost a superset of the capabilities of the "=" and textequ directives. It allows operands that are numeric, text, or string literal constants. The following are all legal uses of the equ directive:

```
One             equ      1
Minus1          equ      -1
TryAgain        equ      'Y'
StringEqu       equ      "Hello there"
TxtEqu          equ      <4[si]>
                  .
                  .
                  .
HTString        byte     StringEqu    ;Same as HTString equ "Hello there"
                  .
                  .
                  .
                mov      ax, TxtEqu   ;Same as mov ax, 4[si]
                  .
                  .
                  .
                mov      bl, One      ;Same as mov bl, 1
                cmp      al, TryAgain ;Same as cmp al, 'Y'
```

Manifest constants you declare with equates help you *parameterize* a program. If you use the same value, string, or text, multiple times within a program, using a symbolic equate will make it very easy to change that value in future modifications to the program. Consider the following example:

```
Array           byte     16 dup (?)
                  .
                  .
                  .
                mov      cx, 16
                mov      bx, 0
ClrLoop:        mov      Array[bx], 0
                inc      bx
                loop     ClrLoop
```

If you decide you want Array to have 32 elements rather than 16, you will need to search throughout your program an locate every reference to this data and adjust the literal constants accordingly. Then there is the possibility that you missed modifying some particular section of code, introducing a bug into your program. On the other hand, if you use the NumElements symbolic constant shown earlier, you would only have to change a single statement in your program, reassemble it, and you would be in business; MASM would automatically update all references using NumElements.

MASM lets you redefine symbols declared with the "=" directive. That is, the following is perfectly legal:

```
SomeSymbol      =        0
                  .
                  .
                  .
SomeSymbol      =        1
```

Since you can change the value of a constant in the program, the symbol's *scope* (where the symbol has a particular value) becomes important. If you could not redefine a symbol, one would expect the symbol to have that constant value everywhere in the program. Given that you can redefine a constant, a symbol's scope cannot be the entire program. The solution MASM uses is the obvious one, a manifest constant's scope is from the point it is defined to the point it is redefined. This has one important ramification – *you must declare all manifest constants with the "=" directive before you use that constant.* Of course, once you redefine a symbolic constant, the previous value of that constant is forgotten. Note that you cannot redefine symbols you declare with the textequ or equ directives.

8.6 Processor Directives

By default, MASM will only assemble instructions that are available on *all* members of the 80x86 family. In particular, this means it will *not* assemble instructions that are not available on the 8086 and 8088 microprocessors. By generating an error for non-8086 instructions, MASM prevents the accidental use of instructions that are not available on various processors. This is great unless, of course, you actually *want* to use those instructions available on processors beyond the 8086 and 8088. The processor directives let you enable the assembly of instructions available on later processors.

The processor directives are

.8086	.8087	.186	.286	.287
.286P	.386	.387	.386P	.486
.486P	.586	.586P		

None of these directives accept any operands.

The processor directives enable all instructions available on a given processor. Since the 80x86 family is upwards compatible, specifying a particular processor directive enables all instructions on that processor and all earlier processors as well.

The .8087, .287, and .387 directives activate the floating point instruction set for the given floating point coprocessors. However, the .8086 directive also enables the 8087 instruction set; likewise, .286 enables the 80287 instruction set and .386 enables the 80387 floating point instruction set. About the only purpose for these FPU (floating point unit) directives is to allow 80287 instructions with the 8086 or 80186 instruction set or 80387 instruction with the 8086, 80186, or 80286 instruction set.

The processor directives ending with a "P" allow assembly of *privileged mode* instructions. Privileged mode instructions are only useful to those writing operating systems, certain device drivers, and other advanced system routines. Since this text does not discuss privileged mode instructions, there is little need to discuss these privileged mode directives further.

The 80386 and later processors support two types of segments when operating in protected mode – 16 bit segments and 32 bit segments. In real mode, these processors support only 16 bit segments. The assembler must generate subtly different opcodes for 16 and 32 bit segments. If you've specified a 32 bit processor using .386, .486, or .586, MASM generates instructions for 32 bit segments by default. If you attempt to run such code in real mode under MS-DOS, you will probably crash the system. There are two solutions to this problem. The first is to specify use16 as an operand to each segment you create in your program. The other solution is slightly more practical, simply put the following statement after the 32 bit processor directive:

```
option    segment:use16
```

This directive tells MASM to generate 16 bit segments by default, rather than 32 bit segments.

Note that MASM does not require an 80486 or Pentium processor if you specify the .486 or .586 directives. The assembler itself is written in 80386 code[3] so you only need an 80386 processor to assemble any program with MASM. Of course, if you use 80486 or Pentium processor specific instructions, you will need an 80486 or Pentium processor to run the assembled code.

You can selectively enable or disable various instruction sets throughout your program. For example, you can turn on 80386 instructions for several lines of code and then return back to 8086 only instructions. The following code sequence demonstrates this:

3. Starting with version 6.1.

```
.386                    ;Begin using 80386 instructions
 .
 .                      ;This code can have 80386 instrs.
 .
.8086                   ;Return back to 8086-only instr set.
 .
 .                      ;This code can only have 8086 instrs.
 .
```

It is possible to write a routine that detects, at run-time, what processor a program is actually running on. Therefore, you can detect an 80386 processor and use 80386 instructions. If you do not detect an 80386 processor, you can stick with 8086 instructions. By selectively turning 80386 instructions on in those sections of your program that executes if an 80386 processor is present, you can take advantage of the additional instructions. Likewise, by turning off the 80386 instruction set in other sections of your program, you can prevent the inadvertent use of 80386 instructions in the 8086-only portion of the program.

8.7 Procedures

Unlike HLLs, MASM doesn't enforce strict rules on exactly what constitutes a procedure[4]. You can call a procedure at any address in memory. The first ret instruction encountered along that execution path terminates the procedure. Such expressive freedom, however, is often abused yielding programs that are very hard to read and maintain. Therefore, MASM provides facilities to declare procedures within your code. The basic mechanism for declaring a procedure is:

```
procname        proc    {NEAR or FAR}
                <statements>
procname        endp
```

As you can see, the definition of a procedure looks similar to that for a segment. One difference is that procname (that is the name of the procedure you're defining) must be a unique identifier within your program. Your code calls this procedure using this name, it wouldn't do to have another procedure by the same name; if you did, how would the program determine which routine to call?

Proc allows several different operands, though we will only consider three: the single keyword near, the single keyword far, or a blank operand field[5]. MASM uses these operands to determine if you're calling this procedure with a near or far call instruction. They also determine which type of ret instruction MASM emits within the procedure. Consider the following two procedures:

```
NProc           proc    near
                mov     ax, 0
                ret
NProc           endp

FProc           proc    far
                mov     ax, 0FFFFH
                ret
FProc           endp
```

and:

```
                call    NPROC
                call    FPROC
```

The assembler automatically generates a three-byte (near) call for the first call instruction above because it knows that NProc is a near procedure. It also generates a five-byte (far) call instruction for the second call because FProc is a far procedure. Within the proce-

4. “Procedure” in this text means any program unit such as procedure, subroutine, subprogram, function, operator, etc.

5. Actually, there are many other possible operands but we will not consider them in this text.

dures themselves, MASM automatically converts all ret instructions to near or far returns depending on the type of routine.

Note that if you do not terminate a proc/endp section with a ret or some other transfer of control instruction and program flow runs into the endp directive, execution will continue with the next executable instruction following the endp. For example, consider the following:

```
Proc1           proc
                mov     ax, 0
Proc1           endp

Proc2           proc
                mov     bx, 0FFFFH
                ret
Proc2           endp
```

If you call Proc1, control will flow on into Proc2 starting with the mov bx,0FFFFh instruction. Unlike high level language procedures, an assembly language procedure does not contain an implicit return instruction before the endp directive. So always be aware of how the proc/endp directives work.

There is nothing special about procedure declarations. They're a convenience provided by the assembler, nothing more. You could write assembly language programs for the rest of your life and never use the proc and endp directives. Doing so, however, would be poor programming practice. Proc and endp are marvelous documentation features which, when properly used, can help make your programs much easier to read and maintain.

MASM versions 6.0 and later treat all statement labels inside a procedure as *local*. That is, you cannot refer directly to those symbols outside the procedure. For more details, see "How to Give a Symbol a Particular Type" on page 385.

8.8 Segments

All programs consist of one or more segments. Of course, while your program is running, the 80x86's segment registers point at the currently active segments. On 80286 and earlier processors, you can have up to four active segments at once (code, data, extra, and stack); on the 80386 and later processors, there are two additional segment registers: fs and gs. Although you cannot access data in more than four or six segments at any one given instant, you can modify the 80x86's segment registers and point them at other segments in memory under program control. This means that a program can access more than four or six segments. The question is "how do you create these different segments in a program and how do you access them at run-time?"

Segments, in your assembly language source file, are defined with the segment and ends directives. You can put as many segments as you like in your program. Well, actually you are limited to 65,536 different segments by the 80x86 processors and MASM probably doesn't even allow that many, but you will probably never exceed the number of segments MASM allows you to put in your program.

When MS-DOS begins execution of your program, it initializes two segment registers. It points cs at the segment containing your main program and it points ss at your stack segment. From that point forward, you are responsible for maintaining the segment registers yourself.

To access data in some particular segment, an 80x86 segment register must contain the address of that segment. If you access data in several different segments, your program will have to load a segment register with that segment's address before accessing it. If you are frequently accessing data in different segments, you will spend considerable time reloading segment registers. Fortunately, most programs exhibit locality of reference when accessing data. This means that a piece of code will likely access the same group of variables many times during a given time period. It is easy to organize your programs so

that variables you often access together appear in the same segment. By arranging your programs in this manner, you can minimize the number of times you need to reload the segment registers. In this sense, a segment is nothing more than a cache of often accessed data.

In real mode, a segment can be up to 64 Kilobytes long. Most pure assembly language programs use less than 64K code, 64K global data, and 64K stack space. Therefore, you can often get by with no more than three or four segments in your programs. In fact, the SHELL.ASM file (containing the skeletal assembly language program) only defines four segments and you will generally only use three of them. If you use the SHELL.ASM file as the basis for your programs, you will rarely need to worry about segmentation on the 80x86. On the other hand, if you want to write complex 80x86 programs, you will need to understand segmentation.

A segment in your file should take the following form[6]:

```
segmentname     segment    {READONLY} {align} {combine} {use} {'class'}
                statements
segmentname     ends
```

The following sections describe each of the operands to the segment directive.

Note: segmentation is a concept that many beginning assembly language programmers find difficult to understand. Note that you do not have to completely understand segmentation to begin writing 80x86 assembly language programs. If you make a copy of the SHELL.ASM file for each program you write, you can effectively ignore segmentation issues. The main purpose of the SHELL.ASM file is to take care of the segmentation details for you. As long as you don't write extremely large programs or use a vast amount of data, you should be able to use SHELL.ASM and forget about segmentation. Nonetheless, eventually you may want to write larger assembly language programs, or you may want to write assembly language subroutines for a high level language like Pascal or C++. At that point you will need to know quite a bit about segmentation. The bottom line is this, you can get by without having to learn about segmentation right now, but sooner or later you will need to understand it if you intend to continue writing 80x86 assembly language code.

8.8.1 Segment Names

The segment directive requires a label in the label field. This label is the segment's name. MASM uses segment names for three purposes: to combine segments, to determine if a *segment override prefix* is necessary, and to obtain the address of a segment. You must also specify the segment's name in the label field of the ends directive that ends the segment.

If the segment name is not unique (i.e., you've defined it somewhere else in the program), the other uses must also be segment definitions. If there is another segment with this same name, then the assembler treats this segment definition as a continuation of the previous segment using the same name. Each segment has its own location counter value associated with it. When you begin a new segment (that is, one whose name has not yet appeared in the source file) MASM creates a new location counter variable, initially zero, for the segment. If MASM encounters a segment definition that is a continuation of a previous segment, then MASM uses the value of the location counter at the end of that previous segment. E.g.,

6. MASM 5.0 and later also provide *simplified segment directives*. In MASM 5.0 they actually were simplified. Since then Microsoft has enhanced them over and over again. Today they are quite complex beasts. They are useful for simplifying the interface between assembly and HLLs. However, we will ignore those directives.

```
CSEG            segment
                mov         ax, bx
                ret
CSEG            ends

DSEG            segment
Item1           byte        0
Item2           word        0
DSEG            ends

CSEG            segment
                mov         ax, 10
                add         ax, Item1
                ret
CSEG            ends
                end
```

The first segment (CSEG) starts with a location counter value of zero. The mov ax,bx instruction is two bytes long and the ret instruction is one byte long, so the location counter is three at the end of the segment. DSEG is another three byte segment, so the location counter associated with DSEG also contains three at the end of the segment. The third segment has the same name as the first segment (CSEG), therefore the assembler will assume that they are the same segment with the second occurrence simply being an extension of the first. Therefore, code placed in the second CSEG segment will be assembled starting at offset three within CSEG – effectively continuing the code in the first CSEG segment.

Whenever you specify a segment name as an operand to an instruction, MASM will use the immediate addressing mode and substitute the address of that segment for its name. Since you cannot load an immediate value into a segment register with a single instruction, loading the segment address into a segment register typically takes two instructions. For example, the following three instructions appear at the beginning of the SHELL.ASM file, they initialize the ds and es registers so they point at the dseg segment:

```
mov     ax, dseg    ;Loads ax with segment address of dseg.
mov     ds, ax      ;Point ds at dseg.
mov     es, ax      ;Point es at dseg.
```

The other purpose for segment names is to provide the segment component of a variable name. Remember, 80x86 addresses contain two components: a segment and an offset. Since the 80x86 hardware defaults most data references to the data segment, it is common practice among assembly language programmers to do the same thing; that is, not bother to specify a segment name when accessing variables in the data segment. In fact, a full variable reference consists of the segment name, a colon, and the offset name:

```
mov     ax, dseg:Item1
mov     dseg:Item2, ax
```

Technically, you should prefix all your variables with the segment name in this fashion. However, most programmers don't bother because of the extra typing involved. Most of the time you can get away with this; however, there are a few times when you really will need to specify the segment name. Fortunately, those situations are rare and only occur in very complex programs, not the kind you're likely to run into for a while.

It is important that you realize that specifying a segment name before a variable's name does not mean that you can access data in a segment without having some segment register pointing at that segment. Except for the jmp and call instructions, there are no 80x86 instructions that let you specify a full 32 bit segmented direct address. All other memory references use a segment register to supply the segment component of the address.

8.8.2 Segment Loading Order

Segments normally load into memory in the order that they appear in your source file. In the example above, DOS would load the CSEG segment into memory *before* the DSEG

segment. Even though the CSEG segment appears in two parts, both before and after DSEG. CSEG's declaration before any occurrence of DSEG tells DOS to load the entire CSEG segment into memory before DSEG. To load DSEG before CSEG, you could use the following program:

```
DSEG            segment  public
DSEG            ends

CSEG            segment  public
                mov        ax, bx
                ret
CSEG            ends

DSEG            segment  public
Item1           byte       0
Item2           word       0
DSEG            ends

CSEG            segment  public
                mov        ax, 10
                add        ax, Item1
                ret
CSEG            ends
                end
```

The empty segment declaration for DSEG doesn't emit any code. The location counter value for DSEG is zero at the end of the segment definition. Hence it's zero at the beginning of the next DSEG segment, exactly as it was in the previous version of this program. However, since the DSEG declaration appears first in the program, DOS will load it into memory first.

The order of appearance is only one of the factors controlling the loading order. For example, if you use the ".alpha" directive, MASM will organize the segments alphabetically rather than in order of first appearance. The optional operands to the segment directive also control segment loading order. These operands are the subject of the next section.

8.8.3 Segment Operands

The segment directive allows six different items in the operand field: an align operand, a combine operand, a class operand, a readonly operand, a "uses" operand, and a size operand. Three of these operands control how DOS loads the segment into memory, the other three control code generation.

8.8.3.1 The ALIGN Type

The align parameter is one of the following words: byte, word, dword, para, or page. These keywords instruct the assembler, linker, and DOS to load the segment on a byte, word, double word, paragraph, or page boundary. The align parameter is optional. If one of the above keywords does not appear as a parameter to the segment directive, the default alignment is paragraph (a paragraph is a multiple of 16 bytes).

Aligning a segment on a byte boundary loads the segment into memory starting at the first available byte after the last segment. Aligning on a word boundary will start the segment at the first byte with an even address after the last segment. Aligning on a dword boundary will locate the current segment at the first address that is an even multiple of four after the last segment.

For example, if segment #1 is declared first in your source file and segment #2 immediate follows and is byte aligned, the segments will be stored in memory as follows (see Figure 8.1).

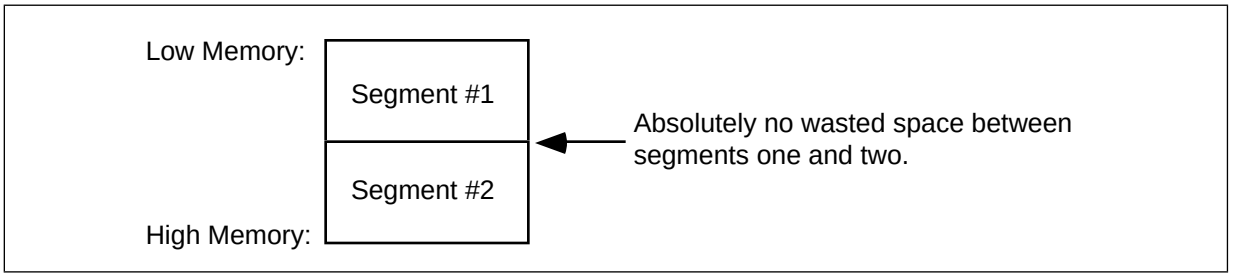


Figure 8.1 Segment with Byte Alignment

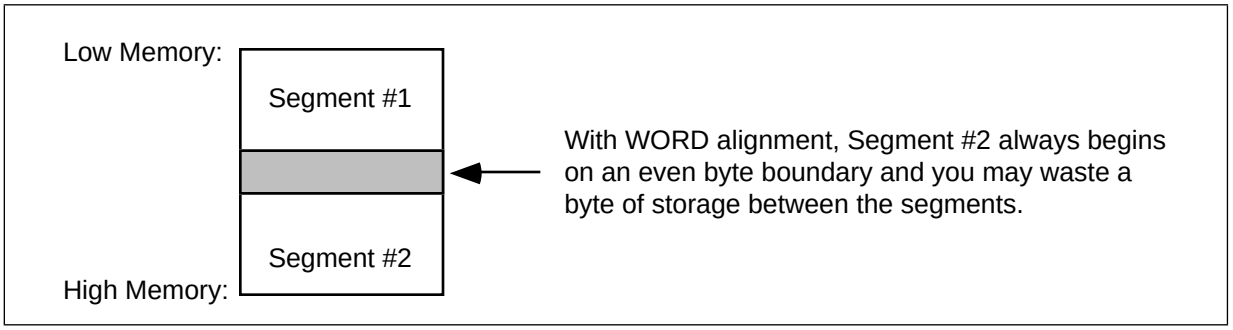


Figure 8.2 Segment with Word Alignment

```
seg1            segment
                  .
                  .
                  .
seg1            ends
seg2            segment   byte
                  .
                  .
                  .
seg2            ends
```

If segments one and two are declared as below, and segment #2 is word aligned, the segments appear in memory as show in Figure 8.2.

```
seg1            segment
                  .
                  .
                  .
seg1            ends

seg2            segment   word
                  .
                  .
                  .
seg2            ends
```

Another example: if segments one and two are as below, and segment #2 is double word aligned, the segments will be stored in memory as shown in Figure 8.3.

```
seg1            segment
                  .
                  .
                  .
seg1            ends

seg2            segment   dword
                  .
                  .
                  .
seg2            ends
```

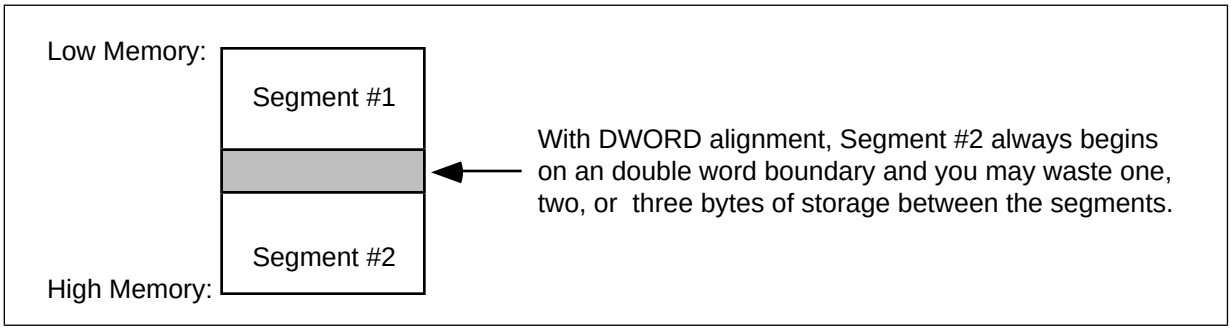


Figure 8.3 Segment with DWord Alignment

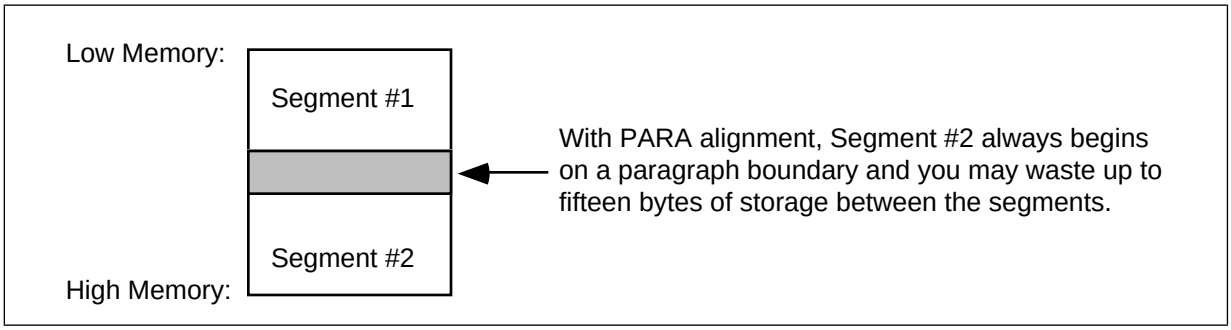


Figure 8.4 Segment with Paragraph Alignment

Since the 80x86's segment registers always point at paragraph addresses, most segments are aligned on a 16 byte paragraph (para) boundary. For the most part, your segments should always be aligned on a paragraph boundary unless you have a good reason to choose otherwise.

For example, if segments one and two are declared as below, and segment #2 is paragraph aligned, DOS will store the segments in memory as shown in Figure 8.4.

```
seg1            segment
                  .
                  .
                  .
seg1            ends

seg2            segment   para
                  .
                  .
                  .
seg2            ends
```

Page boundary alignment forces the segment to begin at the next address that is an even multiple of 256 bytes. Certain data buffers may require alignment on 256 (or 512) byte boundaries. The page alignment option can be useful in this situation.

For example, if segments one and two are declared as below, and segment #2 is page aligned, the segments will be stored in memory as shown in Figure 8.5

```
seg1            segment
                  .
                  .
                  .
seg1            ends

seg2            segment   page
                  .
                  .
                  .
seg2            ends
```

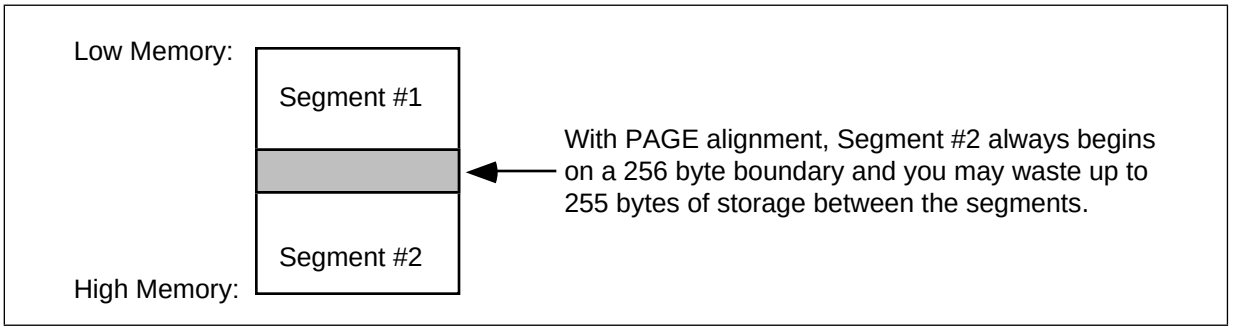


Figure 8.5 Segment with Page Alignment

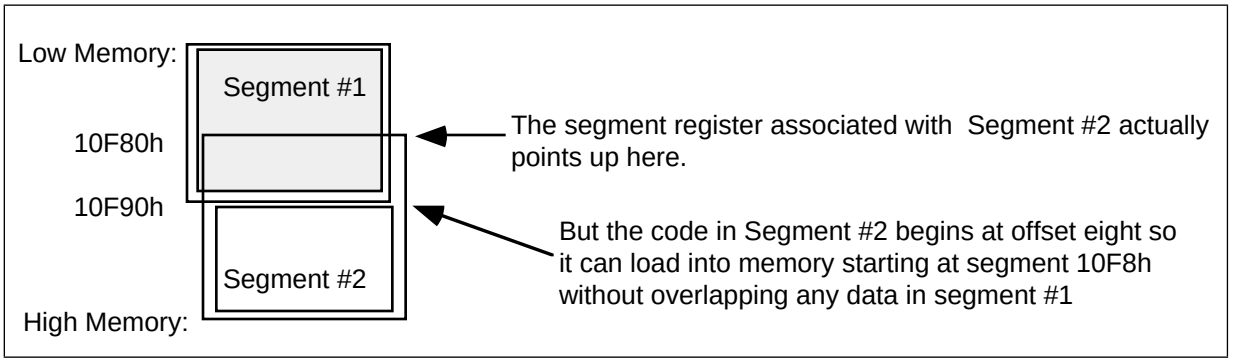


Figure 8.6 Paragraph Alignment of Segments

If you choose any alignment other than byte, the assembler, linker, and DOS may insert several dummy bytes between the two segments, so that the segment is properly aligned. Since the 80x86 segment registers must always point at a paragraph address (that is, they must be paragraph aligned), you might wonder how the processor can address a segment that is aligned on a byte, word, or double word boundary. It's easy. Whenever you specify a segment alignment which forces the segment to begin at an address that is not a paragraph boundary, the assembler/linker will assume that the segment register points at the previous paragraph address and the location counter will begin at some offset into that segment other than zero. For example, suppose that segment #1 above ends at physical address 10F87h and segment #2 is byte aligned. The code for segment #2 will begin at segment address 10F80h. However, this will overlap segment #1 by eight bytes. To overcome this problem, the location counter for segment #2 will begin at 8, so the segment will be loaded into memory just beyond segment #1.

If segment #2 is byte aligned and segment #1 doesn't end at an even paragraph address, MASM adjusts the starting location counter for segment #2 so that it can use the previous paragraph address to access it (see Figure 8.6).

Since the 80x86 requires all segments to start on a paragraph boundary in memory, the Microsoft Assembler (by default) assumes that you want paragraph alignment for your segments. The following segment definition is always aligned on a paragraph boundary:

```
CSEG            segment
                mov         ax, bx
                ret
CSEG            ends
                end
```

8.8.3.2 The COMBINE Type

The combine type controls the order that segments *with the same name* are written out to the object code file produced by the assembler. To specify the combine type you use one of the keywords public, stack, common, memory, or at. Memory is a synonym for public provided for compatibility reasons; you should always use public rather than memory. Common and at are advanced combine types that won't be considered in this text. The stack combine type should be used with your stack segments (see "The SHELL.ASM File" on page 170 for an example). The public combine type should be used with most everything else.

The public and stack combine types essentially perform the same operation. They concatenate segments with the same name into a single contiguous segment, just as described earlier. The difference between the two is the way that DOS handles the initialization of the stack segment and stack pointer registers. All programs should have at least one stack type segment (or the linker will generate a warning); the rest should all be public . MS-DOS will automatically point the stack segment register at the segment you declare with the stack combine type when it loads the program into memory.

If you do not specify a combine type, then the assembler will not concatenate the segments when producing the object code file. In effect, the absence of any combine type keyword produces a *private* combine type by default. Unless the class types are the same (see the next section), each segment will be emitted as MASM encounters it in the source file. For example, consider the following program:

```
CSEG            segment     public
                mov         ax, 0
                mov         VAR1, ax
CSEG            ends

DSEG            segment     public
I               word        ?
DSEG            ends

CSEG            segment     public
                mov         bx, ax
                ret
CSEG            ends

DSEG            segment     public
J               word        ?
DSEG            ends
                end
```

This program section will produce the same code as:

```
CSEG            segment     public
                mov         ax, 0
                mov         VAR1, ax
                mov         bx, ax
                ret
CSEG            ends

DSEG            segment     public
I               word      ?
J               word      ?
DSEG            ends
                end
```

The assembler automatically joins all segments that have the same name and are public. The reason the assembler allows you to separate the segments like this is for convenience. Suppose you have several procedures, each of which requires certain variables. You could declare all the variables in one segment somewhere, but this is often distracting. Most people like to declare their variables right before the procedure that uses them. By using the public combine type with the segment declaration, you may declare your variables right before using them and the assembler will automatically move those variable declarations into the proper segment when assembling the program. For example,

```
CSEG            segment     public

; This is procedure #1

DSEG            segment     public

;Local vars for proc #1.

VAR1            word        ?
DSEG            ends

                mov         AX, 0
                mov         VAR1, AX
                mov         BX, AX
                ret

; This is procedure #2

DSEG            segment     public
I               word        ?
J               word        ?
DSEG            ends

                mov         ax, I
                add         ax, J
                ret
CSEG            ends
                end
```

Note that you can nest segments any way you please. Unfortunately, Microsoft's Macro Assembler scoping rules do not work the same way as a HLL like Pascal. Normally, once you define a symbol within your program, it is visible everywhere else in the program[7].

8.8.4 The CLASS Type

The final operand to the segment directive is usually the class type. The class type specifies the ordering of segments that do not have the same segment name. This operand consists of a symbol enclosed by apostrophes (quotation marks are not allowed here). Generally, you should use the following names: CODE (for segments containing program code); DATA (for segments containing variables, constant data, and tables); CONST (for segments containing constant data and tables); and STACK (for a stack segment). The following program section illustrates their use:

```
CSEG            segment     public 'CODE'
                mov         ax, bx
                ret
CSEG            ends

DSEG            segment     public 'DATA'
Item1           byte    0
Item2           word    0
DSEG            ends

CSEG            segment     public 'CODE'
                mov         ax, 10
                add         AX, Item1
                ret
CSEG            ends

SSEG            segment     stack 'STACK'
STK             word        4000 dup (?)
SSEG            ends

C2SEG           segment     public 'CODE'
                ret
C2SEG           ends
                end
```

7. The major exception are statement labels within a procedure.

The actual loading procedure is accomplished as follows. The assembler locates the first segment in the file. Since it's a public combined segment, MASM concatenates all other CSEG segments to the end of this segment. Finally, since its combine class is 'CODE', MASM appends all segments (C2SEG) with the same class afterwards. After processing these segments, MASM scans the source file for the next uncombined segment and repeats the process. In the example above, the segments will be loaded in the following order: CSEG, CSEG (2nd occurrence), C2SEG, DSEG, and then SSEG. The general rule concerning how your files will be loaded into memory is the following:

- (1) The assembler combines all public segments that have the same name.
- (2) Once combined, the segments are output to the object code file in the order of their appearance in the source file. If a segment name appears twice within a source file (and it's public), then the combined segment will be output to the object code file at the position denoted by the first occurrence of the segment within the source file.
- (3) The linker reads the object code file produced by the assembler and rearranges the segments when creating the executable file. The linker begins by writing the first segment found in the object code file to the .EXE file. Then it searches throughout the object code file for every segment with the same class name. Such segments are sequentially written to the .EXE file.
- (4) Once all the segments with the same class name as the first segment are emitted to the .EXE file, the linker scans the object code file for the next segment which doesn't belong to the same class as the previous segment(s). It writes this segment to the .EXE file and repeats step (3) for each segment belonging to this class.
- (5) Finally, the linker repeats step (4) until it has linked all the segments in the object code file.

8.8.5 The Read-only Operand

If readonly is the first operand of the segment directive, the assembler will generate an error if it encounters any instruction that attempts to write to this segment. This is most useful for code segments, though is it possible to imagine a read-only data segment. This option does not actually *prevent* you from writing to this segment at run-time. It is very easy to trick the assembler and write to this segment anyway. However, by specifying readonly you can catch some common programming errors you would otherwise miss. Since you will rarely place writable variables in your code segments, it's probably a good idea to make your code segments readonly.

Example of READONLY operand:

```
seg1            segment  readonly para public 'DATA'
                  .
                  .
                  .
seg1            ends
```

8.8.6 The USE16, USE32, and FLAT Options

When working with an 80386 or later processor, MASM generates different code for 16 versus 32 bit segments. When writing code to execute in real mode under DOS, you must always use 16 bit segments. Thirty-two bit segments are only applicable to programs running in protected mode. Unfortunately, MASM often defaults to 32 bit mode whenever you select an 80386 or later processor using a directive like .386, .486, or .586 in your program. If you want to use 32 bit instructions, you will have to explicitly tell MASM to use 16 bit segments. The use16, use32, and flat operands to the segment directive let you specify the segment size.

For most DOS programs, you will always want to use the use16 operand. This tells MASM that the segment is a 16 bit segment and it assembles the code accordingly. *If you use one of the directives to activate the 80386 or later instruction sets, you should put use16 in all your code segments or MASM will generate bad code.*

Example of use16 operand:

```
seg1            segment  para public use16 'data'
                  .
                  .
                  .
seg1            ends
```

The use32 and flat operands tell MASM to generate code for a 32 bit segment. Since this text does not deal with protected mode programming, we will not consider these options. See the MASM Programmer's Guide for more details.

If you want to force use16 as the default in a program that allows 80386 or later instructions, there is one way to accomplish this. Place the following directive in your program before any segments:

```
                .option     segment:use16
```

8.8.7 Typical Segment Definitions

Has the discussion above left you totally confused? Don't worry about it. Until you're writing extremely large programs, you needn't concern yourself with all the operands associated with the segment directive. For most programs, the following three segments should prove sufficient:

```
DSEG            segment     para public 'DATA'

; Insert your variable definitions here

DSEG            ends

CSEG            segment     para public use16 'CODE'

; Insert your program instructions here

CSEG            ends

SSEG            segment     para stack 'STACK'
stk             word        1000h dup (?)
EndStk          equ         this word
SSEG            ends
                end
```

The SHELL.ASM file automatically declares these three segments for you. If you always make a copy of the SHELL.ASM file when writing a new assembly language program, you probably won't need to worry about segment declarations and segmentation in general.

8.8.8 Why You Would Want to Control the Loading Order

Certain DOS calls require that you pass the length of your program as a parameter. Unfortunately, computing the length of a program containing several segments is a very difficult process. However, when DOS loads your program into memory, it will load the entire program into a contiguous block of RAM. Therefore, to compute the length of your program, you need only know the starting and ending addresses of your program. By simply taking the difference of these two values, you can compute the length of your program.

In a program that contains multiple segments, you will need to know which segment was loaded first and which was loaded last in order to compute the length of your program. As it turns out, DOS always loads the program segment prefix, or PSP, into mem-

ory just before the first segment of your program. You must consider the length of the PSP when computing the length of your program. MS-DOS passes the segment address of the PSP in the ds register. So computing the difference of the last byte in your program and the PSP will produce the length of your program. The following code segment computes the length of a program in paragraphs:

```
CSEG            segment     public 'CODE'
                mov         ax, ds                ;Get PSP segment address
                sub         ax, seg LASTSEG ;Compute difference

; AX now contains the length of this program (in paragraphs)
                  .
                  .
                  .
CSEG            ends

; Insert ALL your other segments here.

LASTSEG         segment     para public 'LASTSEG'
LASTSEG         ends
                end
```

8.8.9 Segment Prefixes

When the 80x86 references a memory operand, it usually references a location within the current data segment[8]. However, you can instruct the 80x86 microprocessor to reference data in one of the other segments using a segment prefix before an address expression.

A segment prefix is either ds:, cs:, ss:, es:, fs:, or gs:. When used in front of an address expression, a segment prefix instructs the 80x86 to fetch its memory operand from the specified segment rather than the default segment. For example, mov ax, cs:I[bx] loads the accumulator from address I+bx *within the current code segment*. If the cs: prefix were absent, this instruction would normally load the data from the current data segment. Likewise, mov ds:[bp],ax stores the accumulator into the memory location pointed at by the bp register in the current data segment (remember, whenever using bp as a base register it points into the stack segment).

Segment prefixes are instruction opcodes. Therefore, whenever you use a segment prefix you are increasing the length (and decreasing the speed) of the instruction utilizing the segment prefix. Therefore, you don't want to use segment prefixes unless you have a good reason to do so.

8.8.10 Controlling Segments with the ASSUME Directive

The 80x86 generally references data items relative to the ds segment register (or stack segment). Likewise, all code references (jumps, calls, etc.) are always relative to the current code segment. There is only one catch – how does the assembler know which segment is the data segment and which is the code segment (or other segment)? The segment directive doesn't tell you what type of segment it happens to be in the program. Remember, a data segment is a data segment *because the ds register points at it*. Since the ds register can be changed at run time (using an instruction like mov ds,ax), any segment can be a data segment. This has some interesting consequences for the assembler. When you specify a segment in your program, not only must you tell the CPU that a segment is a data segment[9], but you must also tell the assembler where and when that segment is a data (or code / stack / extra / F / G) segment. The assume directive provides this information to the assembler.

8. The exception, of course, are those instructions and addressing modes that use the stack segment by default (e.g., push / pop and addressing modes that use bp or sp).
9. By loading DS with the address of that segment.

The assume directive takes the following form:

```
assume {CS:seg} {DS:seg} {ES:seg} {FS:seg} {GS:seg} {SS:seg}
```

The braces surround optional items, you do not type the braces as part of these operands. Note that there must be at least one operand. Seg is either the name of a segment (defined with the segment directive) or the reserved word nothing. Multiple operands in the operand field of the assume directive must be separated by commas. Examples of valid assume directives:

```
assume      DS:DSEG
assume      CS:CSEG, DS:DSEG, ES:DSEG, SS:SSEG
assume      CS:CSEG, DS:NOTHING
```

The assume directive tells the assembler that you have loaded the specified segment register(s) with the segment addresses of the specified value. **Note that this directive does *not* modify any of the segment registers, it simply tells the assembler to assume the segment registers are pointing at certain segments in the program.** Like the processor selection and equate directives, the assume directive modifies the assembler's behavior from the point MASM encounters it until another assume directive changes the stated assumption.

Consider the following program:

```
DSEG1           segment     para public 'DATA'
var1            word        ?
DSEG1           ends

DSEG2           segment     para public 'DATA'
var2            word        ?
DSEG2           ends

CSEG            segment     para public 'CODE'
                assume      CS:CSEG, DS:DSEG1, ES:DSEG2
                mov         ax, seg DSEG1
                mov         ds, ax
                mov         ax, seg DSEG2
                mov         es, ax

                mov         var1, 0
                mov         var2, 0
                 .
                 .
                 .
                assume      DS:DSEG2
                mov         ax, seg DSEG2
                mov         ds, ax
                mov         var2, 0
                 .
                 .
                 .
CSEG            ends
                end
```

Whenever the assembler encounters a symbolic name, it checks to see which segment contains that symbol. In the program above, var1 appears in the DSEG1 segment and var2 appears in the DSEG2 segment. Remember, the 80x86 microprocessor doesn't know about segments declared within your program, it can only access data in segments pointed at by the cs, ds, es, ss, fs, and gs segment registers. The assume statement in this program tells the assembler the ds register points at DSEG1 for the first part of the program and at DSEG2 for the second part of the program.

When the assembler encounters an instruction of the form mov var1,0, the first thing it does is determine var1's segment. It then compares this segment against the list of assumptions the assembler makes for the segment registers. If you didn't declare var1 in one of these segments, then the assembler generates an error claiming that the program cannot access that variable. If the symbol (var1 in our example) appears in one of the currently assumed segments, then the assembler checks to see if it is the data segment. If so, then the instruction is assembled as described in the appendices. If the symbol appears in

a segment other than the one that the assembler assumes ds points at, then the assembler emits a segment override prefix byte, specifying the actual segment that contains the data.

In the example program above, MASM would assemble mov VAR1,0 without a segment prefix byte. MASM would assemble the first occurrence of the mov VAR2,0 instruction with an es: segment prefix byte since the assembler assumes es, rather than ds, is pointing at segment DSEG2. MASM would assemble the second occurrence of this instruction without the es: segment prefix byte since the assembler, at that point in the source file, assumes that ds points at DSEG2. Keep in mind that it is very easy to confuse the assembler. Consider the following code:

```
CSEG            segment     para public 'CODE'
                assume      CS:CSEG, DS:DSEG1, ES:DSEG2
                mov         ax, seg DSEG1
                mov         ds, ax
                  .
                  .
                  .
                jmp         SkipFixDS

                assume      DS:DSEG2

FixDS:          mov         ax, seg DSEG2
                mov         ds, ax
SkipFixDS:
                  .
                  .
                  .
CSEG            ends
                end
```

Notice that this program jumps around the code that loads the ds register with the segment value for DSEG2. This means that at label SkipFixDS the ds register contains a pointer to DSEG1, not DSEG2. However, the assembler isn't bright enough to realize this problem, so it blindly assumes that ds points at DSEG2 rather than DSEG1. This is a disaster waiting to happen. Because the assembler assumes you're accessing variables in DSEG2 while the ds register actually points at DSEG1, such accesses will reference memory locations in DSEG1 at the same offset as the variables accessed in DSEG2. This will scramble the data in DSEG1 (or cause your program to read incorrect values for the variables assumed to be in segment DSEG2).

For beginning programmers, the best solution to the problem is to avoid using multiple (data) segments within your programs as much as possible. Save the multiple segment accesses for the day when you’re prepared to deal with problems like this. As a beginning assembly language programmer, simply use one code segment, one data segment, and one stack segment and leave the segment registers pointing at each of these segments while your program is executing. The assume directive is quite complex and can get you into a considerable amount of trouble if you misuse it. Better not to bother with fancy uses of assume until you are quite comfortable with the whole idea of assembly language programming and segmentation on the 80x86.

The nothing reserved word tells the assembler that you haven't the slightest idea where a segment register is pointing. It also tells the assembler that you're not going to access any data relative to that segment register unless you explicitly provide a segment prefix to an address. A common programming convention is to place assume directives before all procedures in a program. Since segment pointers to declared segments in a program rarely change except at procedure entry and exit, this is the ideal place to put assume directives:

```
                assume  ds:P1Dseg, cs:cseg, es:nothing
Procedure1      proc    near
                push    ds              ;Preserve DS
                push    ax              ;Preserve AX
                mov     ax, P1Dseg      ;Get pointer to P1Dseg into the
                mov     ds, ax          ; ds register.
                  .
                  .
                  .
                pop     ax              ;Restore ax's value.
                pop     ds              ;Restore ds' value.
                ret
Procedure1      endp
```

The only problem with this code is that MASM still assumes that ds points at P1Dseg when it encounters code after Procedure1. The best solution is to put a second assume directive after the endp directive to tell MASM it doesn't know anything about the value in the ds register:

```
                  .
                  .
                  .
                ret
Procedure1      endp
                assume  ds:nothing
```

Although the next statement in the program will probably be yet another assume directive giving the assembler some new assumptions about ds (at the beginning of the procedure that follows the one above), it's still a good idea to adopt this convention. If you fail to put an assume directive before the next procedure in your source file, the assume ds:nothing statement above will keep the assembler from assuming you can access variables in P1Dseg.

Segment override prefixes always override any assumptions made by the assembler. mov ax, cs:var1 always loads the ax register with the word at offset var1 within the current code segment, regardless of where you've defined var1. The main purpose behind the segment override prefixes is handling indirect references. If you have an instruction of the form mov ax,[bx] the assembler assumes that bx points into the data segment. If you really need to access data in a different segment you can use a segment override, thusly, mov ax, es:[bx].

In general, if you are going to use multiple data segments within your program, you should use full segment:offset names for your variables. E.g., mov ax, DSEG1:I and mov bx,DSEG2:J. This does not eliminate the need to load the segment registers or make proper use of the assume directive, but it will make your program easier to read and help MASM locate possible errors in your program.

The assume directive is actually quite useful for other things besides just setting the default segment. You'll see some more uses for this directive a little later in this chapter.

8.8.11 Combining Segments: The GROUP Directive

Most segments in a typical assembly language program are less than 64 Kilobytes long. Indeed, most segments are *much* smaller than 64 Kilobytes in length. When MS-DOS loads the program's segments into memory, several of the segments may fall into a single 64K region of memory. In practice, you could combine these segments into a single segment in memory. This might possibly improve the efficiency of your code if it saves having to reload segment registers during program execution.

So why not simply combine such segments in your assembly language code? Well, as the next section points out, maintaining separate segments can help you structure your programs better and help make them more modular. This modularity is very important in your programs as they get more complex. As usual, improving the structure and modularity of your programs may cause them to become less efficient. Fortunately, MASM provides a directive, group, that lets you treat two segments as the same physical segment without abandoning the structure and modularity of your program.

The group directive lets you create a new segment name that encompasses the segments it groups together. For example, if you have two segments named "Module1Data" and "Module2Data" that you wish to combine into a single physical segment, you could use the group directive as follows:

```
ModuleData      group   Module1Data, Module2Data
```

The only restriction is that the end of the second module's data must be no more than 64 kilobytes away from the start of the first module in memory. MASM and the linker will *not* automatically combine these segments and place them together in memory. If there are other segments between these two in memory, then the total of all such segments must be less than 64K in length. To reduce this problem, you can use the class operand to the segment directive to tell the linker to combine the two segments in memory by using the same class name:

```
ModuleData      group   Module1Data, Module2Data

Module1Data     segment para public 'MODULES'
                  .
                  .
                  .
Module1Data     ends
                  .
                  .
                  .
Module2Data     segment byte public 'MODULES'
                  .
                  .
                  .
Module2Data     ends
```

With declarations like those above, you can use "ModuleData" anywhere MASM allows a segment name, as the operand to a mov instruction, as an operand to the assume directive, etc. The following example demonstrates the usage of the ModuleData segment name:

```
                assume  ds:ModuleData
Module1Proc     proc    near
                push    ds                      ;Preserve ds' value.
                push    ax                      ;Preserve ax's value.
                mov     ax, ModuleData          ;Load ds with the segment
address
                mov     ds, ax                  ; of ModuleData.
                  .
                  .
                  .
                pop     ax                      ;Restore ax's and ds' values.
                pop     ds
                ret
Module1Proc     endp
                assume  ds:nothing
```

Of course, using the group directive in this manner hasn't really improved the code. Indeed, by using a different name for the data segment, one could argue that using group in this manner has actually obfuscated the code. However, suppose you had a code sequence that needed to access variables in both the Module1Data and Module2Data segments. If these segments were physically and logically separate you would have to load two segment registers with the addresses of these two segments in order to access their data concurrently. This would cost you a segment override prefix on all the instructions that access one of the segments. If you cannot spare an extra segment register, the situation will be even worse, you'll have to constantly load new values into a single segment register as you access data in the two segments. You can avoid this overhead by combining the two logical segments into a single physical segment and accessing them through their group rather than individual segment names.

If you group two or more segments together, all you're really doing is creating a pseudo-segment that encompasses the segments appearing in the group directive's operand field. Grouping segments does not prevent you from accessing the individual segments in the grouping list. The following code is perfectly legal:

```
                assume  ds:Module1Data
                mov     ax, Module1Data
                mov     ds, ax
                  .
        < Code that accesses data in Module1Data >
                  .
                assume  ds:Module2Data
                mov     ax, Module2Data
                mov     ds, ax
                  .
        < Code that accesses data in Module2Data >
                  .
                assume  ds:ModuleData
                mov     ax, ModuleData
                mov     ds, ax
                  .
        < Code that accesses data in both Module1Data and Module2Data >
                  .
                  .
                  .
```

When the assembler processes segments, it usually starts the location counter value for a given segment at zero. Once you group a set of segments, however, an ambiguity arises; grouping two segments causes MASM and the linker to concatenate the variables of one or more segments to the end of the first segment in the group list. They accomplish this by adjusting the offsets of all symbols in the concatenated segments as though they were all symbols in the same segment. The ambiguity exists because MASM allows you to reference a symbol in its segment or in the group segment. The symbol has a different offset depending on the choice of segment. To resolve the ambiguity, MASM uses the following algorithm:

- If MASM doesn't know that a segment register is pointing at the symbol's segment or a group containing that segment, MASM generates an error.
- If an assume directive associates the segment name with a segment register but does not associate a segment register with the group name, then MASM uses the offset of the symbol within its segment.
- If an assume directive associates the group name with a segment register but does not associate a segment register with the symbol's segment name, MASM uses the offset of the symbol with the group.
- If an assume directive provides segment register association with both the symbol's segment and its group, MASM will pick the offset that would not require a segment override prefix. For example, if the assume directive specifies that ds points at the group name and es points at the segment name, MASM will use the group offset if the default segment register would be ds since this would not require MASM to emit a segment override prefix opcode. If either choice results in the emission of a segment override prefix, MASM will choose the offset (and segment override prefix) associated with the symbol's segment.

MASM uses the algorithm above if you specify a variable name without a segment prefix. If you specify a segment register override prefix, then MASM may choose an arbitrary offset. Often, this turns out to be the group offset. So the following instruction sequence, without an assume directive telling MASM that the BadOffset symbol is in seg1 may produce bad object code:

```
DataSegs        group   Data1, Data2, Data3
                  .
                  .
                  .
Data2           segment
                  .
                  .
                  .
BadOffset       word    ?
                  .
                  .
                  .
Data2           ends
                  .
                  .
                  .
```

```
                assume  ds:nothing, es:nothing, fs:nothing, gs:nothing
                mov     ax, Data2               ;Force ds to point at data2
despite
                mov     ds, ax                  ; the assume directive above.

                mov     ax, ds:BadOffset        ;May use the offset from
DataSegs
                                                ; rather than Data2!
```

If you want to force the correct offset, use the variable name containing the complete segment:offset address form:

```
; To force the use of the offset within the DataSegs group use an instruction
; like the following:

                mov     ax, DataSegs:BadOffset

; To force the use of the offset within Data2, use:

                mov     ax, Data2:BadOffset
```

You must use extra care when working with groups within your assembly language programs. If you force MASM to use an offset within some particular segment (or group) and the segment register is not pointing at that particular segment or group, MASM may not generate an error message and the program will not execute correctly. Reading the offsets MASM prints in the assembly listing will not help you find this error. MASM *always* displays the offsets within the symbol's segment in the assembly listing. The only way to really detect that MASM and the linker are using bad offsets is to get into a debugger like CodeView and look at the actual machine code bytes produced by the linker and loader.

8.8.12 Why Even Bother With Segments?

After reading the previous sections, you're probably wondering what possible good could come from using segments in your programs. To be perfectly frank, if you use the SHELL.ASM file as a skeleton for the assembly language programs you write, you can get by quite easily without ever worrying about segments, groups, segment override prefixes, and full segment:offset names. As a beginning assembly language programmer, it's probably a good idea to ignore much of this discussion on segmentation until you are much more comfortable with 80x86 assembly language programming. However, there are three reasons you'll want to learn more about segmentation if you continue writing assembly language programs for any length of time: the real-mode 64K segment limitation, program modularity, and interfacing with high level languages.

When operating in real mode, segments can be a maximum of 64 kilobytes long. If you need to access more than 64K of data or code in your programs, you will need to use more than one segment. This fact, more than any other reason, has dragged programmers (kicking and screaming) into the world of segmentation. Unfortunately, this is as far as many programmers get with segmentation. They rarely learn more than just enough about segmentation to write a program that accesses more than 64K of data. As a result, when a segmentation problem occurs because they don't fully understand the concept, they blame segmentation for their problems and they avoid using segmentation as much as possible.

This is too bad because segmentation is a powerful memory management tool that lets you organize your programs into logical entities (*segments*) that are, in theory, independent of one another. The field of software engineering studies how to write correct, large programs. Modularity and independence are two of the primary tools software engineers use to write large programs that are correct and easy to maintain. The 80x86 family provides, in hardware, the tools to implement segmentation. On other processors, segmentation is enforced strictly by software. As a result, it is easier to work with segments on the 80x86 processors.

Although this text does not deal with protected mode programming, it is worth pointing out that when you operate in protected mode on 80286 and later processors, the 80x86 hardware can actually prevent one module from accessing another module's data (indeed, the term "protected mode" means that segments are protected from illegal access). Many debuggers available for MS-DOS operate in protected mode allowing you to catch array and segment bounds violations. Soft-ICE and Bounds Checker from NuMega are examples of such products. Most people who have worked with segmentation in a protected mode environment (e.g., OS/2 or Windows) appreciate the benefits that segmentation offers.

Another reason for studying segmentation on the 80x86 is because you might want to write an assembly language function that a high level language program can call. Since the HLL compiler makes certain assumptions about the organization of segments in memory, you will need to know a little bit about segmentation in order to write such code.

8.9 The END Directive

The end directive terminates an assembly language source file. In addition to telling MASM that it has reached the end of an assembly language source file, the end directive's optional operand tells MS-DOS where to transfer control when the program begins execution; that is, you specify the name of the main procedure as an operand to the end directive. If the end directive's operand is not present, MS-DOS will begin execution starting at the first byte in the .exe file. Since it is often inconvenient to guarantee that your main program begins with the first byte of object code in the .exe file, most programs specify a starting location as the operand to the end directive. If you are using the SHELL.ASM file as a skeleton for your assembly language programs, you will notice that the end directive already specifies the procedure main as the starting point for the program.

If you are using separate assembly and you're linking together several different object code files (see "Managing Large Programs" on page 425), only one module can have a main program. Likewise, only one module should specify the starting location of the program. If you specify more than one starting location, you will confuse the linker and it will generate an error.

8.10 Variables

Global variable declarations use the byte/sbyte/db, word/sword/dw, dword/sdword/dd, qword/dq, and tbyte/dt pseudo-opcodes. Although you can place your variables in any segment (including the code segment), most beginning assembly language programmers place all their global variables in a single data segment..

A typical variable declaration takes the form:

```
varname          byte     initial_value
```

Varname is the name of the variable you're declaring and initial_value is the initial value you want that variable to have when the program begins execution. "?" is a special initial value. It means that you don't want to give a variable an initial value. When DOS loads a program containing such a variable into memory, it does not initialize this variable to any particular value.

The declaration above reserves storage for a single byte. This could be changed to any other variable type by simply changing the byte mnemonic to some other appropriate pseudo-opcode.

For the most part, this text will assume that you declare all variables in a *data segment*, that is, a segment that the 80x86's ds register will point at. In particular, most of the programs herein will place all variables in the DSEG segment (CSEG is for code, DSEG is for data, and SSEG is for the stack). See the SHELL.ASM program in Chapter Four for more details on these segments.

Since Chapter Five covers the declaration of variables, data types, structures, arrays, and pointers in depth, this chapter will not waste any more time discussing this subject. Refer to Chapter Five for more details.

8.11 Label Types

One unusual feature of Intel syntax assemblers (like MASM) is that they are *strongly typed*. A strongly typed assembler associates a certain type with symbols declared appearing in the source file and will generate a warning or an error message if you attempt to use that symbol in a context that doesn't allow its particular type. Although unusual in an assembler, most high level languages apply certain typing rules to symbols declared in the source file. Pascal, of course, is famous for being a strongly typed language. You cannot, in Pascal, assign a string to a numeric variable or attempt to assign an integer value to a procedure label. Intel, in designing the syntax for 8086 assembly language, decided that all the reasons for using a strongly typed language apply to assembly language as well as Pascal. Therefore, standard Intel syntax 80x86 assemblers, like MASM, impose certain type restrictions on the use of symbols within your assembly language programs.

8.11.1 How to Give a Symbol a Particular Type

Symbols, in an 80x86 assembly language program, may be one of eight different primitive types: byte, word, dword, qword, tbyte, near, far, and abs (constant)[10]. Anytime you define a label with the byte, word, dword, qword, or tbyte pseudo-opcodes, MASM associates the type of that pseudo-opcode with the label. For example, the following variable declaration will create a symbol of type byte:

```
BVar            byte    ?
```

Likewise, the following defines a dword symbol:

```
DWVar           dword   ?
```

Variable types are not limited to the primitive types built into MASM. If you create your own types using the typedef or struct directives MASM will associate those types with any associated variable declarations.

You can define near symbols (also known as statement labels) in a couple of different ways. First, all procedure symbols declared with the proc directive (with either a blank operand field[11] or near in the operand field) are near symbols. Statement labels are also near symbols. A statement label takes the following form:

```
label:          instr
```

Instr represents an 80x86 instruction[12]. Note that a colon must follow the symbol. It is not part of the symbol, the colon informs the assembler that this symbol is a statement label and should be treated as a near typed symbol.

Statement labels are often the targets of jump and loop instructions. For example, consider the following code sequence:

```
                mov     cx, 25
Loop1:          mov     ax, cx
                call    PrintInteger
                loop    Loop1
```

10. MASM also supports an FWORD type. FWORD is for programmers working in 32-bit protected mode. This text will not consider that type.

11. Note: if you are using the simplified directives, a blank operand field might not necessarily imply that the procedure is near. If your program does not contain a ".MODEL" directive, however, blank operand fields imply a near type.

12. The mnemonic "instr" is optional. You may also place a statement label on a line by itself. The assembler assigns the location counter of the next instruction in the program to the symbol.

The loop instruction decrements the cx register and transfers control to the instruction labelled by Loop1 until cx becomes zero.

Inside a procedure, statement labels are *local*. That is, the scope of statement labels inside a procedure are visible only to code inside that procedure. If you want to make a symbol global to a procedure, place two colons after the symbol name. In the example above, if you needed to refer to Loop1 outside of the enclosing procedure, you would use the code:

```
                mov     cx, 25
Loop1::         mov     ax, cx
                call    PrintInteger
                loop    Loop1
```

Generally, far symbols are the targets of jump and call instructions. The most common method programmers use to create a far label is to place far in the operand field of a proc directive. Symbols that are simply constants are normally defined with the equ directive. You can also declare symbols with different types using the equ and extrn/extern/externdef directives. An explanation of the extrn directives appears in the section "Managing Large Programs" on page 425.

If you declare a numeric constant using an equate, MASM assigns the type *abs* (absolute, or constant) to the system. Text and string equates are given the type *text*. You can also assign an arbitrary type to a symbol using the equ directive, see "Type Operators" on page 392 for more details.

8.11.2 Label Values

Whenever you define a label using a directive or pseudo-opcode, MASM gives it a type and a value. The value MASM gives the label is usually the current location counter value. If you define the symbol with an equate the equate's operand usually specifies the symbol's value. When encountering the label in an operand field, as with the loop instruction above, MASM substitutes the label's value for the label.

8.11.3 Type Conflicts

Since the 80x86 supports strongly typed symbols, the next question to ask is "What are they used for?" In a nutshell, strongly typed symbols can help verify proper operation of your assembly language programs. Consider the following code sections:

```
DSEG            segment     public 'DATA'
                  .
                  .
                  .
I               byte        ?
                  .
                  .
                  .
DSEG            ends

CSEG            segment     public 'CODE'
                  .
                  .
                  .
                mov       ax, I
                  .
                  .
                  .
CSEG            ends
                end
```

The mov instruction in this example is attempting to load the ax register (16 bits) from a byte sized variable. Now the 80x86 microprocessor is perfectly capable of this operation. It would load the al register from the memory location associated with I and load the ah register from the next successive memory location (which is probably the L.O. byte of some other variable). However, this probably wasn't the original intent. The person who

wrote this code probably forgot that I is a byte sized variable and assumed that it was a word variable – which is definitely an error in the logic of the program.

MASM would never allow an instruction like the one above to be assembled without generating a diagnostic message. This can help you find errors in your programs, particularly difficult-to-find errors. On occasion, advanced assembly language programmers may want to execute a statement like the one above. MASM provides certain coercion operators that bypass MASM's safety mechanisms and allow illegal operations (see “Coercion” on page 390).

8.12 Address Expressions

An *address expression* is an algebraic expression that produces a numeric result that MASM merges into the displacement field of an instruction. An integer constant is probably the simplest example of an address expression. The assembler simply substitutes the value of the numeric constant for the specified operand. For example, the following instruction fills the immediate data fields of the mov instruction with zeros:

```
mov     ax, 0
```

Another simple form of an addressing mode is a symbol. Upon encountering a symbol, MASM substitutes the value of that symbol. For example, the following two statements emit the same object code as the instruction above:

```
Value           equ     0
                mov     ax, Value
```

An address expression, however, can be much more complex than this. You can use various arithmetic and logical operators to modify the basic value of some symbols or constants.

Keep in mind that MASM computes address expressions during assembly, not at run time. For example, the following instruction does not load ax from location Var and add one to it:

```
mov     ax, Var1+1
```

Instead, this instruction loads the al register with the byte stored at the address of Var1 plus one and then loads the ah register with the byte stored at the address of Var1 plus two.

Beginning assembly language programmers often confuse computations done at assembly time with those done at run time. Take extra care to remember that MASM computes all address expressions at assembly time!

8.12.1 Symbol Types and Addressing Modes

Consider the following instruction:

```
jmp     Location
```

Depending on how the label Location is defined, this jmp instruction will perform one of several different operations. If you'll look back at the chapter on the 80x86 instruction set, you'll notice that the jmp instruction takes several forms. As a recap, they are

```
jmp     label       (short)
jmp     label       (near)
jmp     label       (far)
jmp     reg         (indirect near, through register)
jmp     mem/reg     (indirect near, through memory)
jmp     mem/reg     (indirect far, thorugh memory)
```

Notice that MASM uses the same mnemonic (jmp) for each of these instructions; how does it tell them apart? The secret lies with the operand. If the operand is a statement label within the current segment, the assembler selects one of the first two forms depend-

ing on the distance to the target instruction. If the operand is a statement label within a different segment, then the assembler selects jmp (far) label. If the operand following the jmp instruction is a register, then MASM uses the indirect near jmp and the program jumps to the address in the register. If a memory location is selected, the assembler uses one of the following jumps:

- NEAR if the variable was declared with word/sword/dw
- FAR if the variable was declared with dword/sdword/dd

An error results if you've used byte/sbyte/db, qword/dq, or tbyte/dt or some other type.

If you've specified an indirect address, e.g., jmp [bx], the assembler will generate an error because it cannot determine if bx is pointing at a word or a dword variable. For details on how you specify the size, see the section on coercion in this chapter.

8.12.2 Arithmetic and Logical Operators

MASM recognizes several arithmetic and logical operators. The following tables provide a list of such operators:

Table 36: Arithmetic Operators

Operator	Syntax	Description
+	*+expr*	Positive (unary)
-	*-expr*	Negation (unary)
+	*expr* + *expr*	Addition
-	*expr* - *expr*	Subtraction
*	*expr* * *expr*	Multiplication
/	*expr* / *expr*	Division
MOD	*expr* MOD *expr*	Modulo (remainder)
[ ]	*expr* [*expr*]	Addition (index operator)

Table 37: Logical Operators

Operator	Syntax	Description
SHR	expr SHR expr	Shift right
SHL	expr SHL expr	Shift left
NOT	NOT expr	Logical (bit by bit) NOT
AND	expr AND expr	Logical AND
OR	expr OR expr	Logical OR
XOR	expr XOR expr	Logical XOR

Table 38: Relational Operators

Operator	Syntax	Description
EQ	*expr* EQ *expr*	True (0FFh) if equal, false (0) otherwise
NE	*expr* NE *expr*	True (0FFh) if not equal, false (0) otherwise
LT	*expr* LT *expr*	True (0FFh) if less, false (0) otherwise
LE	*expr* LE *expr*	True (0FFh) if less or equal, false (0) otherwise
GT	*expr* GT *expr*	True (0FFh) if greater, false (0) otherwise
GE	*expr* GE *expr*	True (0FFh) if greater or equal, false (0) otherwise

You must not confuse these operators with 80x86 instructions! The addition operator adds two values together, their sum becomes an operand to an instruction. This addition is performed when assembling the program, not at run time. If you need to perform an addition at execution time, use the add or adc instructions.

You're probably wondering "What are these operators used for?" The truth is, not much. The addition operator gets used quite a bit, the subtraction somewhat, the comparisons once in a while, and the rest even less. Since addition and subtraction are the only operators beginning assembly language programmers regularly employ, this discussion considers only those two operators and brings up the others as required throughout this text.

The addition operator takes two forms: expr+expr or expr[expr]. For example, the following instruction loads the accumulator, not from memory location COUNT, but from the very next location in memory:

```
mov     al, COUNT+1
```

The assembler, upon encountering this statement, will compute the sum of COUNT's address plus one. The resulting value is the memory address for this instruction. As you may recall, the mov al, memory instruction is three bytes long and takes the form:

Opcode | L. O. Displacement Byte | H. O. Displacement Byte

The two displacement bytes of this instruction contain the sum COUNT+1.

The expr[expr] form of the addition operation is for accessing elements of arrays. If AryData is a symbol that represents the address of the first element of an array, AryData[5] represents the address of the fifth byte into AryData. The expression AryData+5 produces the same result, and either could be used interchangeably, however, for arrays the expr[expr] form is a little more self documenting. One trap to avoid: expr1[expr2][expr3] does not automatically index (properly) into a two dimensional array for you. This simply computes the sum expr1+expr2+expr3.

The subtraction operator works just like the addition operator, except it computes the difference rather than the sum. This operator will become very important when we deal with local variables in Chapter 11.

Take care when using multiple symbols in an address expression. MASM restricts the operations you can perform on symbols to addition and subtraction and only allows the following forms:

Expression:	Resulting type:
reloc + const	Reloc, at address specified.
reloc - const	Reloc, at address specified.
reloc - reloc	Constant whose value is the number of bytes between the first and second operands. Both variables must physically appear in the same segment in the current source file.

Reloc stands for *relocatable symbol or expression*. This can be a variable name, a statement label, a procedure name, or any other symbol associated with a memory location in the program. It could also be an expression that produces a relocatable result. MASM does not allow any operations other than addition and subtraction on expressions whose resulting type is relocatable. You cannot, for example, compute the product of two relocatable symbols.

The first two forms above are very common in assembly language programs. Such an address expression will often consist of a single relocatable symbol and a single constant (e.g., "var + 1"). You won't use the third form very often, but it is very useful once in a while. You can use this form of an address expression to compute the distance, in bytes, between two points in your program. The procsize symbol in the following code, for example, computes the size of Proc1:

```
Proc1           proc    near
                push    ax
                push    bx
                push    cx
                mov     cx, 10
                lea     bx, SomeArray
                mov     ax, 0
ClrArray:       mov     [bx], ax
                add     bx, 2
                loop    ClrArray
                pop     cx
                pop     bx
                pop     ax
                ret
Proc1           endp

procsize        =       $ - Proc1
```

"$" is a special symbol MASM uses to denote the current offset within the segment (i.e., the location counter). It is a relocatable symbol, as is Proc1, so the equate above computes the difference between the offset at the start of Proc1 and the end of Proc1. This is the length of the Proc1 procedure, in bytes.

The operands to the operators other than addition and subtraction must be constants or an expression yielding a constant (e.g., "$-Proc1" above produces a constant value). You'll mainly use these operators in macros and with the conditional assembly directives.

8.12.3 Coercion

Consider the following program segment:

```
DSEG            segment     public 'DATA'
I               byte        ?
J               byte        ?
DSEG            ends

CSEG            segment
                .
                .
                .
                mov         al, I
                mov         ah, J
                .
                .
                .
CSEG            ends
```

Since I and J are adjacent, there is no need to use two mov instructions to load al and ah, a simple mov ax, I instruction would do the same thing. Unfortunately, the assembler will balk at mov ax, I since I is a byte. The assembler will complain if you attempt to treat it as a word. As you can see, however, there are times when you'd probably like to treat a byte variable as a word (or treat a word as a byte or double word, or treat a double word as a something else).

Temporarily changing the type of a label for some particular occurrence is *coercion*. Expressions can be coerced to a different type using the MASM ptr operator. You use the ptr operator as follows:

type PTR *expression*

Type is any of byte, word, dword, tbyte, near, far, or other type and *expression* is any general expression that is the address of some object. The coercion operator returns an expression with the same value as *expression*, but with the type specified by *type*. To handle the above problem you'd use the assembly language instruction:

```
                mov     ax, word ptr I
```

This instructs the assembler to emit the code that will load the ax register with the word at address I. This will, of course, load al with I and ah with J.

Code that uses double word values often makes extensive use of the coercion operator. Since lds and les are the only 32-bit instructions on pre-80386 processors, you cannot (without coercion) store an integer value into a 32-bit variable using the mov instruction on those earlier CPUs. If you've declared DBL using the dword pseudo-opcode, then an instruction of the form mov DBL,ax will generate an error because it's attempting to move a 16 bit quantity into a 32 bit variable. Storing values into a double word variable requires the use of the ptr operator. The following code demonstrates how to store the ds and bx registers into the double word variable DBL:

```
mov     word ptr DBL, bx
mov     word ptr DBL+2, ds
```

You will use this technique often as various UCR Standard Library and MS-DOS calls return a double word value in a pair of registers.

Warning: If you coerce a jmp instruction to perform a far jump to a near label, other than performance degradation (the far jmp takes longer to execute), your program will work fine. If you coerce a call to perform a far call to a near subroutine, you're headed for trouble. Remember, far calls push the cs register onto the stack (with the return address). When executing a near ret instruction, the old cs value will not be popped off the stack, leaving junk on the stack. The very next pop or ret instruction will not operate properly since it will pop the cs value off the stack rather than the original value pushed onto the stack[13].

Expression coercion can come in handy at times. Other times it is essential. However, you shouldn't get carried away with coercion since data type checking is a powerful debugging tool built in to MASM. By using coercion, you override this protection provided by the assembler. Therefore, always take care when overriding symbol types with the ptr operator.

One place where you'll need coercion is with the mov memory, immediate instruction. Consider the following instruction:

```
mov     [bx], 5
```

Unfortunately, the assembler has no way of telling whether bx points at a byte, word, or double word item in memory[14]. The value of the immediate operand isn't of any use. Even though five is a byte quantity, this instruction might be storing the value 0005h into a word variable, or 00000005 into a double word variable. If you attempt to assemble this statement, the assembler will generate an error to the effect that you must specify the size of the memory operand. You can easily accomplish this using the byte ptr, word ptr, and dword ptr operators as follows:

```
mov     byte ptr [bx], 5        ;For a byte variable
mov     word ptr [bx], 5        ;For a word variable
mov     dword ptr [bx], 5       ;For a dword variable
```

Lazy programmers might complain that typing strings like “word ptr” or “far ptr” is too much work. Wouldn’t it have been nice had Intel chosen a single character symbol rather than these long phrases? Well, quit complaining and remember the textequ directive. With the equate directive you can substitute a long string like “word ptr” for a short symbol. You’ll find equates like the following in many programs, including several in this text:

```
byp             textequ  <byte ptr>   ;Remember, “bp” is a reserved symbol!
wp              textequ  <word ptr>
dp              textequ  <dword ptr>
np              textequ  <near ptr>
fp              textequ  <far ptr>
```

With equates like the above, you can use statements like the following:

13. The situation when you force a near call to a far procedure is even worse. See the exercises for more details.

14. Actually, you can use the assume directive to tell MASM what bx is pointing at. See the MASM reference manuals for details.

```
mov     byp [bx], 5
mov     ax, wp I
mov     wp DBL, bx
mov     wp DBL+2, ds
```

8.12.4 Type Operators

The “xxxx ptr” coercion operator is an example of a type operator. MASM expressions possess two major attributes: a value and a type. The arithmetic, logical, and relational operators change an expression's value. The type operators change its type. The previous section demonstrated how the ptr operator could change an expression's type. There are several additional type operators as well.

Table 39: Type Operators

Operator	Syntax	Description
PTR	byte ptr *expr* word ptr *expr* dword ptr *expr* qword ptr *expr* tbyte ptr *expr* near ptr *expr* far ptr *expr*	Coerce *expr* to point at a byte. Coerce *expr* to point at a word. Coerce *expr* to point at a dword. Coerce *expr* to point at a qword. Coerce *expr* to point at a tbyte. Coerce *expr* to a near value. Coerce *expr* to a far value.
short	short *expr*	*expr* must be within ±128 bytes of the current jmp instruction (typically a JMP instruction). This operator forces the JMP instruction to be two bytes long (if possible).
this	this *type*	Returns an expression of the specified type whose value is the current location counter.
seg	seg *label*	Returns the segment address portion of *label*.
offset	offset *label*	Returns the offset address portion of *label*.
.type	type *label*	Returns a byte that indicates whether this symbol is a variable, statement label, or structure name. Superceded by opattr.
opattr	opattr *label*	Returns a 16 bit value that gives information about *label*.
length	length *variable*	Returns the number of array elements for a single dimension array. If a multi-dimension array, this operator returns the number of elements for the first dimension.
lengthof	lengthof *variable*	Returns the number of items in array *variable*.
type	type *symbol*	Returns a expression whose type is the same as *symbol* and whose value is the size, in bytes, for the specified symbol.
size	size *variable*	Returns the number of bytes allocated for single dimension array *variable*. Useless for multi-dimension arrays. Superceded by sizeof.
sizeof	sizeof *variable*	Returns the size, in bytes, of array *variable*.
low	low *expr*	Returns the L.O. byte of *expr*.
lowword	lowword *expr*	Returns the L.O. word of *expr*.
high	high *expr*	Returns the H.O. byte of *expr*.
highword	highword *expr*	Returns the H.O. word of *expr*.

The short operator works exclusively with the jmp instruction. Remember, there are two jmp direct near instructions, one that has a range of 128 bytes around the jmp, one that has a range of 32,768 bytes around the current instruction. MASM will automatically generate a short jump if the target address is up to 128 bytes before the current instruction. This operator is mainly present for compatibility with old MASM (pre-6.0) code.

The this operator forms an expression with the specified type whose value is the current location counter. The instruction mov bx, this word, for example, will load the bx register with the value 8B1Eh, the opcode for mov bx, memory. The address this word is the address of the opcode for this very instruction! You mostly use the this operator with the equ directive to give a symbol some type other than constant. For example, consider the following statement:

```
HERE            equ     this near
```

This statement assigns the current location counter value to HERE and sets the type of HERE to near. This, of course, could have been done much easier by simply placing the label HERE: on the line by itself. However, the this operator with the equ directive does have some useful applications, consider the following:

```
WArray          equ     this word
BArray          byte    200 dup (?)
```

In this example the symbol BArray is of type byte. Therefore, instructions accessing BArray must contain byte operands throughout. MASM would flag a mov ax, BArray+8 instruction as an error. However, using the symbol WArray lets you access the same exact memory locations (since WArray has the value of the location counter immediately before encountering the byte pseudo-opcode) so mov ax,WArray+8 accesses location BArray+8. Note that the following two instructions are identical:

```
        mov     ax, word ptr BArray+8
        mov     ax, WArray+8
```

The seg operator does two things. First, it extracts the segment portion of the specified address, second, it converts the type of the specified expression from address to constant. An instruction of the form mov ax, seg symbol always loads the accumulator with the constant corresponding to the segment portion of the address of symbol. If the symbol is the name of a segment, MASM will automatically substitute the paragraph address of the segment for the name. However, it is perfectly legal to use the seg operator as well. The following two statements are identical if dseg is the name of a segment:

```
        mov     ax, dseg
        mov     ax, seg dseg
```

Offset works like seg, except it returns the offset portion of the specified expression rather than the segment portion. If VAR1 is a word variable, mov ax, VAR1 will always load the two bytes at the address specified by VAR1 into the ax register. The mov ax, offset VAR1 instruction, on the other hand, loads the offset (address) of VAR1 into the ax register. Note that you can use the lea instruction or the mov instruction with the offset operator to load the address of a scalar variable into a 16 bit register. The following two instructions both load bx with the address of variable J:

```
        mov     bx, offset J
        lea     bx, J
```

The lea instruction is more flexible since you can specify any memory addressing mode, the offset operator only allows a single symbol (i.e., displacement only addressing). Most programmers use the mov form for scalar variables and the lea instructor for other addressing modes. This is because the mov instruction was faster on earlier processors.

One very common use for the seg and offset operators is to initialize a segment and pointer register with the segmented address of some object. For example, to load es:di with the address of SomeVar, you could use the following code:

```
        mov     di, seg SomeVar
        mov     es, di
        mov     di, offset SomeVar
```

Since you cannot load a constant directly into a segment register, the code above copies the segment portion of the address into di and then copies di into es before copying the offset into di. This code uses the di register to copy the segment portion of the address into es so that it will affect as few other registers as possible.

Opattr returns a 16 bit value providing specific information about the expression that follows it. The .type operator is an older version of opattr that returns the L.O. eight bits of this value. Each bit in the value of these operators has the following meaning:

Table 40: OPATTR/.TYPE Return Value

Bit(s)	Meaning
0	References a label in the code segment if set.
1	References a memory variable or relocatable data object if set.
2	Is an immediate (absolute/constant) value if set.
3	Uses direct memory addressing if set.
4	Is a register name, if set.
5	References no undefined symbols and there is no error, if set.
6	Is an SS: relative reference, if set.
7	References an external name.
8-10	000 - no language type 001 - C/C++ language type 010 - SYSCALL language type 011 - STDCALL language type 100 - Pascal language type 101 - FORTRAN language type 110 - BASIC language type

The language bits are for programmers writing code that interfaces with high level languages like C++ or Pascal. Such programs use the simplified segment directives and MASM's HLL features.

You would normally use these values with MASM's conditional assembly directives and macros. This allows you to generate different instruction sequences depending on the type of a macro parameter or the current assembly configuration. For more details, see "Conditional Assembly" on page 397 and "Macros" on page 400.

The size, sizeof, length, and lengthof operators compute the sizes of variables (including arrays) and return that size and their value. You shouldn't normally use size and length. The sizeof and lengthof operators have superceded these operators. Size and length do not always return reasonable values for arbitrary operands. MASM 6.x includes them to remain compatible with older versions of the assembler. However, you will see an example later in this chapter where you can use these operators.

The sizeof *variable* operator returns the number of bytes directly allocated to the specified variable. The following examples illustrate the point:

```
a1              byte    ?                           ;SIZEOF(a1) = 1
a2              word    ?                           ;SIZEOF(a2) = 2
a4              dword   ?                           ;SIZEOF(a4) = 4
a8              real8   ?                           ;SIZEOF(a8) = 8
ary0            byte    10 dup (0)                  ;SIZEOF(ary0) = 10
ary1            word    10 dup (10 dup (0))         ;SIZEOF(ary1) = 200
```

You can also use the sizeof operator to compute the size, in bytes, of a structure or other data type. This is *very* useful for computing an index into an array using the formula from Chapter Four:

```
Element_Address := base_address + index*Element_Size
```

You may obtain the element size of an array or structure using the sizeof operator. So if you have an array of structures, you can compute an index into the array as follows:

```
        .286                            ;Allow 80286 instructions.
s       struct
        <some number of fields>
s       ends

          .
          .
          .
array   s         16 dup ({})           ;An array of 16 “s” elements

          .
          .
          .
        imul      bx, I, sizeof s       ;Compute BX := I * elementsize
        mov       al, array[bx].fieldname
```

You can also apply the sizeof operator to other data types to obtain their size in bytes. For example, sizeof byte returns 1, sizeof word returns two, and sizeof dword returns 4. Of course, applying this operator to MASM’s built-in data types is questionable since the size of those objects is fixed. However, if you create your own data types using typedef, it makes perfect sense to compute the size of the object using the sizeof operator:

```
integer           typedef  word
Array             integer  16 dup (?)
                    .
                    .
                    .
                  imul     bx, bx, sizeof integer
                    .
                    .
                    .
```

In the code above, sizeof integer would return two, just like sizeof word. However, if you change the typedef statement so that integer is a dword rather than a word, the sizeof integer operand would automatically change its value to four to reflect the new size of an integer.

The lengthof operator returns the total number of elements in an array. For the Array variable above, lengthof Array would return 16. If you have a two dimensional array, lengthof returns the total number of elements in that array.

When you use the lengthof and sizeof operators with arrays, you must keep in mind that it is possible for you to declare arrays in ways that MASM can misinterpret. For example, the following statements all declare arrays containing eight words:

```
A1                word     8 dup (?)

A2                word     1, 2, 3, 4, 5, 6, 7, 8

; Note: the “\” is a “line continuation” symbol. It tells MASM to append
;       the next line to the end of the current line.

A3                word     1, 2, 3, 4, \
                           5, 6, 7, 8

A4                word     1, 2, 3, 4
                  word     5, 6, 7, 8
```

Applying the sizeof and lengthof operators to A1, A2, and A3 produces sixteen (sizeof) and eight (lengthof). However, sizeof(A4) produces eight and lengthof(A4) produces four. This happens because MASM thinks that the arrays begin and end with a single data declaration. Although the A4 declaration sets aside eight consecutive words, just like the other three declarations above, MASM thinks that the two word directives declare two separate arrays rather than a single array. So if you want to initialize the elements of a large array or a multidimensional array and you also want to be able to apply the lengthof and sizeof operators to that array, you should use A3’s form of declaration rather than A4’s.

The type operator returns a constant that is the number of bytes of the specified operand. For example, type(word) returns the value two. This revelation, by itself, isn’t particularly interesting since the size and sizeof operators also return this value. However, when you use the type operator with the comparison operators (eq, ne, le, lt, gt, and ge), the comparison produces a true result only if the types of the operands are the same. Consider the following definitions:

```
Integer         typedef  word
J               word     ?
K               sword    ?
L               integer  ?
M               word     ?

                byte     type (J) eq word              ;value = 0FFh
                byte     type (J) eq sword             ;value = 0
                byte     type (J) eq type (L)          ;value = 0FFh
                byte     type (J) eq type (M)          ;value = 0FFh
                byte     type (L) eq integer           ;value = 0FFh
                byte     type (K) eq dword             ;value = 0
```

Since the code above typedef'd Integer to word, MASM treats integers and words as the same type. Note that with the exception of the last example above, the value on either side of the eq operator is two. Therefore, when using the comparison operations with the type operator, MASM compares more than just the value. Therefore, type and sizeof are not synonymous. E.g.,

```
                byte     type (J) eq type (K)          ;value = 0
                byte     (sizeof J) equ (sizeof K)     ;value = 0FFh
```

The type operator is especially useful when using MASM's conditional assembly directives. See "Conditional Assembly" on page 397 for more details.

The examples above also demonstrate another interesting MASM feature. If you use a type name within an expression, MASM treats it as though you'd entered "type(name)" where *name* is a symbol of the given type. In particular, specifying a type name returns the size, in bytes, of an object of that type. Consider the following examples:

```
Integer         typedef  word
s               struct
d               dword    ?
w               word     ?
b               byte     ?
s               ends

                byte     word                          ;value = 2
                byte     sword                         ;value = 2
                byte     byte                          ;value = 1
                byte     dword                         ;value = 4
                byte     s                             ;value = 7
                byte     word eq word                  ;value = 0FFh
                byte     word eq sword                 ;value = 0
                byte     b eq dword                    ;value = 0
                byte     s eq byte                     ;value = 0
                byte     word eq Integer               ;value = 0FFh
```

The high and low operators, like offset and seg, change the type of expression from whatever it was to a constant. These operators also affect the value of the expression – they decompose it into a high order byte and a low order byte. The high operator extracts bits eight through fifteen of the expression, the low operator extracts and returns bits zero through seven. Highword and lowword extract the H.O. and L.O. 16 bits of an expression (see Figure 8.7).

You can extract bits 16-23 and 24-31 using expressions of the form low(highword(*expr*)) and high(highword(*expr*))[15], respectively.

8.12.5 Operator Precedence

Although you will rarely need to use a complex address expression employing more than two operands and a single operator, the need does arise on occasion. MASM supports a simple operator precedence convention based on the following rules:

- MASM executes operators of a higher precedence first.

15. The parentheses make this expression more readable, they are not required.

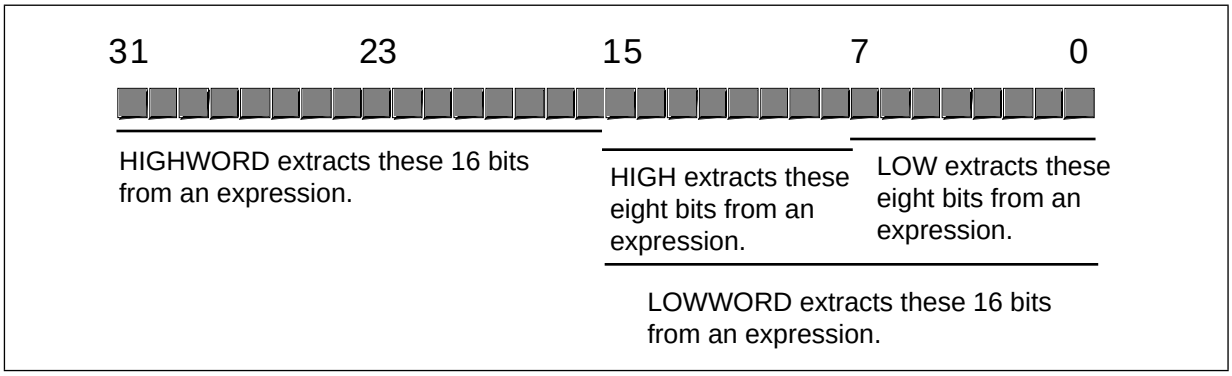


Figure 8.7 HIGHWORD, LOWWORD, HIGH, and LOW Operators

- Operators of an equal precedence are left associative and evaluate from left to right.
- Parentheses override the normal precedence.

Table 41: Operator Precedence

Precedence	Operators
(Highest)	
1	length, lengthof, size, sizeof, ( ), [ ], < >
2	. (structure field name operator)
3	CS: DS: ES: FS: GS: SS: (Segment override prefixes)
4	ptr offset set type opattr this
5	high, low, highword, lowword
6	+ - (unary)
7	* / mod shl shr
8	+ - (binary)
9	eq ne lt le gt ge
10	not
11	and
12	or xor
13	short .type
(Lowest)	

Parentheses should only surround expressions. Some operators, like sizeof and lengthof, require type names, not expressions. They do not allow you to put parentheses around the name. Therefore, "(sizeof X)" is legal, but "sizeof(X)" is not. Keep this in mind when using parentheses to override operator precedence in an expression. If MASM generates an error, you may need to rearrange the parentheses in your expression.

As is true for expressions in a high level language, it is a good idea to always use parentheses to explicitly state the precedence in all complex address expressions (complex meaning that the expression has more than one operator). This generally makes the expression more readable and helps avoid precedence related bugs.

8.13 Conditional Assembly

MASM provides a very powerful conditional assembly facility. With conditional assembly, you can decide, based on certain conditions, whether MASM will assemble the code. There are several conditional assembly directives, the following section covers most of them.

It is important that you realize that these directives evaluate their expressions at *assembly time*, not at run time. The if conditional assembly directive is not the same as a Pascal or C "if" statement. If you are familiar with C, the #ifdef directive in C is roughly equivalent to some of MASM's conditional assembly directives.

MASM's conditional assembly directives are important because they let you generate different object code for different operating environments and different situations. For example, suppose you want to write a program that will run on all machines but you would like to optimize the code for 80386 and later processors. Obviously, you cannot execute 80386 code on an 8086 processor, so how can you solve this problem?

One possible solution is to determine the processor type at run time and execute different sections of code in the program depending on the presence or absence of a 386 or later CPU. The problem with this approach is that your program needs to contain two code sequences – an optimal 80386 sequence and a compatible 8086 sequence. On any given system the CPU will only execute one of these code sequences in the program, so the other sequence will be wasting memory and may have adverse affects on any cache in the system.

A second possibility is to write two versions of the code, one that uses only 8086 instructions and one that uses the full 80386 instruction set. During installation, the user (or the installation program) selects the 80386 version if they have an 80386 or later processor. Otherwise they select the 8086 version. While this marginally increases the cost of the software since it will require more disk space, the program will consume less memory while running. The problem with this approach is that you will need to maintain *two* separate versions of the program. If you correct a bug in the 8086 version of the code, you will probably need to correct that same bug in the 80386 program. Maintaining multiple source files is a difficult task.

A third solution is to use *conditional assembly*. With conditional assembly, you can merge the 8086 and 80386 versions of the code into the same source file. During assembly, you can *conditionally* choose whether MASM assembles the 8086 or the 80386 version. By assembling the code twice, you can produce an 8086 and an 80386 version of the code. Since both versions of the code appear in the same source file, the program will be much easier to maintain since you will not have to correct the same bug in two separate source files. You *may* need to correct the same bug twice in two separate code sequences in the program, but generally the bug will appear in two adjacent code sequences, so it is less likely that you will forget to make the change in both places.

MASM's conditional assembly directives are especially useful within *macros*. They can help you produce efficient code when a macro would normally produce sub-optimal code. For more information about macros and how you can use conditional assembly within a macro, see "Macros" on page 400.

Macros and conditional assembly actually provide "a programming language within a programming language." Macros and conditional assembly let you write programs (in the "macro language") that write segments of assembly language code for you. This introduces an independent way to generate bugs in your application programs. Not only can a bug develop in your assembly language code, you can also introduce bugs in your macro code (e.g., conditional assembly), that wind up producing bugs in your assembly language code. Keep in mind that if you get too sophisticated when using conditional assembly, you can produce programs that are very difficult to read, understand, and debug.

8.13.1 IF Directive

The if directive uses the following syntax:

```
            if       expression
   <sequence of statements>
            else     ;This is optional!
   <sequence of statements>
            endif
```

MASM evaluates *expression*. If it is a non-zero value, then MASM will assemble the statements between the if and else directives (or endif, if the else isn't present). If the expression evaluates to zero (false) and an else section is present, MASM will assemble the statements between the else directive and the endif directive. If the else section is not present and expression evaluates to false, then MASM will not assemble any of the code between the if and endif directives.

The important thing to remember is that *expression* has to be an expression that MASM can evaluate at assembly time. That is, it must evaluate to a constant. Manifest constants (equates) and values that MASM's type operators produce are commonly found in if directive expressions. For example, suppose you want to assemble code for two different processors as described above. You could use statements like the following:

```
Processor       =       80386       ;Set to 8086 for 8086-only code
                .
                .
                .
                if      Processor eq 80386
                shl     ax, 4
                else                ;Must be 8086 processor.
                mov     cl, 4
                shl     ax, cl
                endif
```

There are other ways to accomplish this same thing. MASM provides built-in variables that tell you if you are assembling code for some specific processor. More on that later.

8.13.2 IFE directive

The ife directive is used exactly like the if directive, except it assembles the code after the ife directive only if the expression evaluates to zero (false), rather than true (non-zero).

8.13.3 IFDEF and IFNDEF

These two directives require a single symbol as the operand. Ifdef will assemble the associated code if the symbol is defined, Ifndef will assemble the associated code if the symbol isn't defined. Use else and endif to terminate the conditional assembly sequences.

These directives are especially popular for including or not including code in an assembly language program to handle certain special cases. For example, you could use statements like the following to include debugging statements in your code:

```
ifdef   DEBUG

<place debugging statements here>

endif
```

To activate the debugging code, simply define the symbol DEBUG somewhere at the beginning of your program (before the first ifdef referencing DEBUG). To automatically eliminate the debugging code, simply delete the definition of DEBUG. You may define DEBUG using a simple statement like:

```
DEBUG           =       0
```

Note that the value you assign to DEBUG is unimportant. Only the fact that you have defined (or have not defined) this symbol is important.

8.13.4 IFB, IFNB

These directives, useful mainly in macros (see "Macros" on page 400) check to see if an operand is blank (ifb) or not blank (ifnb). Consider the following code:

```
Blank           textequ     <>
NotBlank        textequ     <not blank>

                ifb         Blank
                <this code will assemble>
                endif

                ifb         NotBlank
                <this code will not>
                endif
```

The ifnb works in an opposite manner to ifb. That is, it would assemble the statements above that ifb does not and vice versa.

8.13.5 IFIDN, IFDIF, IFIDNI, and IFDIFI

These conditional assembly directives take two operands and process the associated code if the operands are identical (ifidn), different (ifdif), identical ignoring case (ifidni), or different ignoring case (ifdifi). The syntax is

```
                ifidn       op1, op2
    <statements to assemble if <op1> = <op2>>
                endif

                ifdif       op1, op2
    <statements to assemble if <op1> ≠ <op2>>
                endif

                ifidni      op1, op2
    <statements to assemble if <op1> = <op2>>
                endif

                ifdifi      op1, op2
    <statements to assemble if <op1> ≠ <op2>>
                endif
```

The difference between the IFxxx and IFxxxI statements above is that the IFxxxI statements ignore differences in alphabetic case when comparing operands.

8.14 Macros

A macro is like a procedure that inserts a block of statements at various points in your program during assembly. There are three general types of macros that MASM supports: procedural macros, functional macros, and looping macros. Along with conditional assembly, these tools provide the traditional if, loop, procedure, and function constructs found in many high level languages. Unlike the assembly instructions you write, the conditional assembly and macro language constructs execute *during assembly*. The conditional assembly and macros statements do not exist when your assembly language program is running. The purpose of these statements is to control which statements MASM assembles into your final ".exe" file. While the conditional assembly directives select or omit certain statements for assembly, the macro directives let you emit repetitive sequences of instructions to an assembly language file like high level language procedures and loops let you repetitively execute sequences of high level language statements.

8.14.1 Procedural Macros

The following sequence defines a macro:

```
name            macro       {parameter1 {parameter2 {,...}}}
            <statements>
                endm
```

Name must be a valid and unique symbol in the source file. You will use this identifier to invoke the macro. The (optional) parameter names are placeholders for values you specify when you invoke the macro; the braces above denote the optional items, they should not actually appear in your source code. These parameter names are local to the macro and may appear elsewhere in the program.

Example of a macro definition:

```
COPY            macro       Dest, Source
                mov         ax, Source
                mov         Dest, ax
                endm
```

This macro will copy the word at the source address to the word at the destination address. The symbols Dest and Source are local to the macro and may appear elsewhere in the program.

Note that MASM does not immediately assemble the instructions between the macro and endm directives when MASM encounters the macro. Instead, the assembler stores the text corresponding to the macro into a special table (called the symbol table). MASM inserts these instructions into your program when you invoke the macro.

To invoke (use) a macro, simply specify the macro name as a MASM mnemonic. When you do this, MASM will insert the statements between the macro and endm directives into your code at the point of the macro invocation. If your macro has parameters, MASM will substitute the actual parameters appearing as operands for the formal parameters appearing in the macro definition. MASM does a straight textual substitution, just as though you had created text equates for the parameters.

Consider the following code that uses the COPY macro defined above:

```
                call        SetUpX
                copy        Y, X
                add         Y, 5
```

This program segment will issue a call to SetUpX (which, presumably, does something to the variable X) then invokes the COPY macro, that copies the value in the variable X into the variable Y. Finally, it adds five to the value contained in variable Y.

Note that this instruction sequence is *absolutely* identical to:

```
                call        SetUpX
                mov         ax, X
                mov         Y, ax
                add         Y, 5
```

In some instances using macros can save a considerable amount of typing in your programs. For example, suppose you want to access elements of various two dimensional arrays. As you may recall, the formula to compute the row-major address for an array element is

element address = base address + (First Index * Row Size + Second Index) * element size

Suppose you want write some assembly code that achieves the same result as the following C code:

```
int a[16][7], b[16][7], x[7][16];
int i,j;

       for (i=0; i<16; i = i + 1)
                 for (j=0; j < 7; j = j + 1)
                          x[j][i] = a[i][j]*b[15-i][j];
```

The 80x86 code for this sequence is rather complex because of the number of array accesses. The complete code is

```
                .386                    ;Uses some 286 & 386 instrs.
                option  segment:use16;Required for real mode programs
                  .
                  .
                  .
a               sword   16 dup (7 dup (?))
b               sword   16 dup (7 dup (?))
x               sword   7 dup (16 dup (?))
                  .
                  .
                  .
i               textequ <cx>            ;Hold I in CX register.
j               textequ <dx>            ;Hold J in DX register.

                mov     I, 0            ;Initialize I loop index with zero.
ForILp:         cmp     I, 16           ;Is I less than 16?
                jnl     ForIDone        ;If so, fall into body of I loop.

                mov     J, 0            ;Initialize J loop index with zero.
ForJLp:         cmp     J, 7            ;Is J less than 7?
                jnl     ForJDone        ;If so, fall into body of J loop.

                imul    bx, I, 7        ;Compute index for a[i][j].
                add     bx, J
                add     bx, bx          ;Element size is two bytes.
                mov     ax, A[bx]       ;Get a[i][j]

                mov     bx, 15          ;Compute index for b[15-I][j].
                sub     bx, I
                imul    bx, 7
                add     bx, J
                add     bx, bx          ;Element size is two bytes.
                imul    ax, b[bx]       ;Compute a[i][j] * b[16-i][j]

                imul    bx, J, 16       ;Compute index for X[J][I]
                add     bx, I
                add     bx, bx
                mov     X[bx], ax       ;Store away result.

                inc     J               ;Next loop iteration.
                jmp     ForJLp

ForJDone:       inc     I               ;Next I loop iteration.
                jmp     ForILp

ForIDone:                               ;Done with nested loop.
```

This is a lot of code for only five C/C++ statements! If you take a close look at this code, you'll notice that a large number of the statements simply compute the index into the three arrays. Furthermore, the code sequences that compute these array indices are very similar. If they were exactly the same, it would be obvious we could write a macro to replace the three array index computations. Since these index computations are *not* identical, one might wonder if it is possible to create a macro that will simplify this code. The answer is yes; by using macro parameters it is very easy to write such a macro. Consider the following code:

```
i               textequ <cx>            ;Hold I in CX register.
j               textequ <dx>            ;Hold J in DX register.

NDX2            macro   Index1, Index2, RowSize
                imul    bx, Index1, RowSize
                add     bx, Index2
                add     bx, bx
                endm

                mov     I, 0            ;Initialize I loop index with zero.
ForILp:         cmp     I, 16           ;Is I less than 16?
                jnl     ForIDone        ;If so, fall into body of I loop.

                mov     J, 0            ;Initialize J loop index with zero.
ForJLp:         cmp     J, 7            ;Is J less than 7?
                jnl     ForJDone        ;If so, fall into body of J loop.

                NDX2    I, J, 7
                mov     ax, A[bx]       ;Get a[i][j]
```

```
                mov     bx, 15      ;Compute index for b[15-I][j].
                sub     bx, I
                NDX2    bx, J, 7
                imul    ax, b[bx]   ;Compute a[i][j] * b[15-i][j]

                NDX2    J, I, 16
                mov     X[bx], ax   ;Store away result.

                inc     J           ;Next loop iteration.
                jmp     ForJLp

ForJDone:       inc     I           ;Next I loop iteration.
                jmp     ForILp

ForIDone:                           ;Done with nested loop.
```

One problem with the NDX2 macro is that you need to know the row size of an array (since it is a macro parameter). In a short example like this one, that isn't much of a problem. However, if you write a large program you can easily forget the sizes and have to look them up or, worse yet, "remember" them incorrectly and introduce a bug into your program. One reasonable question to ask is if MASM could figure out the row size of the array automatically. The answer is yes.

MASM's length operator is a holdover from the pre-6.0 days. It was supposed to return the number of elements in an array. However, all it really returns is the first value appearing in the array's operand field. For example, (length a) would return 16 given the definition for a above. MASM corrected this problem by introducing the lengthof operator that properly returns the total number of elements in an array. (Lengthof a), for example, properly returns 112 (16 * 7). Although the (length a) operator returns the wrong value for our purposes (it returns the column size rather than the row size), we can use its return value to compute the row size using the expression (lengthof a)/(length a). With this knowledge, consider the following two macros:

```
; LDAX-This macro loads ax with the word at address Array[Index1][Index2]
;       Assumptions:        You've declared the array using a statement like
;                             Array word Colsize dup (RowSize dup (?))
;                           and the array is stored in row major order.
;
;       If you specify the (optional) fourth parameter, it is an 80x86
;       machine instruction to substitute for the MOV instruction that
;       loads AX from Array[bx].

LDAX            macro   Array, Index1, Index2, Instr
                imul    bx, Index1, (lengthof Array) / (length Array)
                add     bx, Index2
                add     bx, bx

; See if the caller has supplied the fourth operand.

                ifb     <Instr>
                mov     ax, Array[bx]           ;If not, emit a MOV instr.
                else
                instr   ax, Array[bx]           ;If so, emit user instr.
                endif
                endm

; STAX-This macro stores ax into the word at address Array[Index1][Index2]
;       Assumptions:        Same as above

STAX            macro   Array, Index1, Index2
                imul    bx, Index1, (lengthof Array) / (length Array)
                add     bx, Index2
                add     bx, bx
                mov     Array[bx], ax
                endm
```

With the macros above, the original program becomes:

```
i               textequ  <cx>                ;Hold I in CX register.
j               textequ  <dx>                ;Hold J in DX register.

                mov      I, 0                ;Initialize I loop index with
zero.
ForILp:         cmp      I, 16               ;Is I less than 16?
                jnl      ForIDone            ;If so, fall into body of I
loop.

                mov      J, 0                ;Initialize J loop index with
zero.
ForJLp:         cmp      J, 7                ;Is J less than 7?
                jnl      ForJDone            ;If so, fall into body of J
loop.

                ldax     A, I, J             ;Fetch A[I][J]
                mov      bx, 16              ;Compute 16-I.
                sub      bx, I
                ldax     b, bx, J, imul      ;Multiply in B[16-I][J].
                stax     x, J, I             ;Store to X[J][I]

                inc      J                   ;Next loop iteration.
                jmp      ForJLp

ForJDone:       inc      I                   ;Next I loop iteration.
                jmp      ForILp

ForIDone:                                    ;Done with nested loop.
```

As you can plainly see, the code for the loops above is getting shorter and shorter by using these macros. Of course, the *entire* code sequence is actually *longer* because the macros represent more lines of code that they save in the original program. However, that is an artifact of this particular program. In general, you'd probably have more than three array accesses; furthermore, you can always put the LDAX and STAX macros in a library file and automatically include them anytime you're dealing with two dimensional arrays. Although, technically, your program might actually contain more assembly language statements if you include these macros in your code, *you* only had to write those macros once. After that, it takes very little effort to include the macros in any new program.

We can shorten this code sequence even more using some additional macros. However, there are a few additional topics to cover before we can do that, so keep reading.

8.14.2 Macros vs. 80x86 Procedures

Beginning assembly language programmers often confuse macros and procedures. A procedure is a single section of code that you call from various points in the program. A macro is a sequence of instructions that MASM replicates in your program each time you use the macro. Consider the following two code fragments:

```
Proc_1          proc     near
                mov      ax, 0
                mov      bx, ax
                mov      cx, 5
                ret
Proc_1          endp

Macro_1         macro
                mov      ax, 0
                mov      bx, ax
                mov      cx, 5
                endm

                call     Proc_1
                    .
                    .
                call     Proc_1
                    .
                    .
                Macro_1
                    .
                    .
                Macro_1
```

Although the macro and procedure produce the same result, they do it in different ways. The procedure definition generates code when the assembler encounters the proc directive. A call to this procedure requires only three bytes. At execution time, the 80x86:

- encounters the call instruction,
- pushes the return address onto the stack,
- jumps to Proc_1,
- executes the code therein,
- pops the return address off the stack, and
- returns to the calling code.

The macro, on the other hand, does not emit any code when processing the statements between the macro and endm directives. However, upon encountering Macro_1 in the mnemonic field, MASM will assemble every statement between the macro and endm directives and emit that code to the output file. At run time, the CPU executes these instructions without the call/ret overhead.

The execution of a macro expansion is usually faster than the execution of the same code implemented with a procedure. However, this is another example of the classic speed/space trade-off. Macros execute faster by eliminating the call/return sequence. However, the assembler copies the macro code into your program at each macro invocation. If you have a lot of macro invocations within your program, it will be much larger than the same program that uses procedures.

Macro invocations and procedure invocations are considerably different. To invoke a macro, you simply specify the macro name as though it were an instruction or directive. To invoke a procedure you need to use the call instruction. In many contexts it is unfortunate that you use two separate invocation mechanisms for such similar operations. The real problem occurs if you want to switch a macro to a procedure or vice versa. It might be that you've been using macro expansion for a particular operation, but now you've expanded the macro so many times it makes more sense to use a procedure. Maybe just the opposite is true, you've been using a procedure but you want to expand the code in-line to improve it's performance. The problem with either conversion is that you will have to find every invocation of the macro or procedure call and modify it. Modifying the procedure or macro is easy, but locating and changing all the invocations can be quite a bit of work. Fortunately, there is a very simple technique you can use so procedure calls share the same syntax as macro invocation. The trick is to create a macro or a text equate for each procedure you write that expands into a call to that procedure. For example, suppose you write a procedure ClearArray that zeros out arrays. When writing the code, you could do the following:

```
ClearArray      textequ  <call $$ClearArray>
$$ClearArray    proc     near
                  .
                  .
                  .
$$ClearArray    endm
```

To call the ClearArray procedure, you'd simply use a statement like the following:

```
                  .
                  .
                  .
        <Set up parameters for ClearArray>
                 ClearArray
                  .
                  .
                  .
```

If you ever change the $$ClearArray procedure to a macro, all you need to do is name it ClearArray and dispose of the textequ for the procedure. Conversely, if you already have a macro and you want to convert it to a procedure, Simply name the procedure $$*procname* and create a text equate that emits a call to this procedure. This allows you to use the same invocation syntax for procedures or macros.

This text won't normally use the technique described above, except for the UCR Standard Library routines. This is not because this isn't a good way to invoke procedures. Some people have trouble differentiating macros and procedures, so this text will use

explicit calls to help avoid that confusion. Standard Library calls are an exception because using macro invocations is the standard way to call these routines.

8.14.3 The LOCAL Directive

Consider the following macro definition:

```
LJE             macro     Dest
                jne       SkipIt
                jmp       Dest
SkipIt:
                endm
```

This macro does a "long jump if equal". However, there is one problem with it. Since MASM copies the macro text verbatim (allowing, of course, for parameter substitution), the symbol SkipIt will be redefined each time the LJE macro appears. When this happens, the assembler will generate a multiple definition error. To overcome this problem, the local directive can be used to define a *local* symbol within the macro. Consider the following macro definition:

```
LJE             macro       Dest
                local       SkipIt
                jne         SkipIt
                jmp         Dest
SkipIt:
                endm
```

In this macro definition, SkipIt is a local symbol. Therefore, the assembler will generate a new copy of SkipIt each time you invoke the macro. This will prevent MASM from generating an error.

The local directive, if it appears within your macro definition, must appear immediately after the macro directive. If you need multiple local symbols, you can specify several of them in the local directive's operand field. Simply separate each symbol with a comma:

```
IFEQUAL         macro     a, b
                local     ElsePortion, Done
                mov       ax, a
                cmp       ax, b
                jne       ElsePortion
                inc       bx
                jmp       Done
ElsePortion:    dec       bx
Done:
                endm
```

8.14.4 The EXITM Directive

The exitm directive immediately terminates the expansion of a macro, exactly as though MASM encountered endm. MASM ignores all text from the exitm directive to the endm.

You're probably wondering why anyone would ever use the exitm directive. After all, if MASM ignores all text between exitm and endm, why bother sticking an exitm directive into your macro in the first place? The answer is conditional assembly. Conditional assembly can be used to conditionally execute the exitm directive, thereby allowing further macro expansion under certain conditions, consider the following:

```
Bytes           macro       Count
                byte        Count
                if          Count eq 0
                exitm
                endif
                byte        Count dup (?)
                endm
```

Of course, this simple example could have been coded without using the exitm directive (the conditional assembly directive is all we require), but it does demonstrate how the exitm directive can be used within a conditional assembly sequence to control its influence.

8.14.5 Macro Parameter Expansion and Macro Operators

Since MASM does a textual substitution for macro parameters when you invoke a macro, there are times when a macro invocation might not produce the results you expect. For example, consider the following (admittedly dumb) macro definition:

```
Index           =       8

; Problem-      This macro attempts to load AX with the element of a word
;               array specified by the macro's parameter. This parameter
;               must be an assembly-time constant.

Problem         macro   Parameter
                mov     ax, Array[Parameter*2]
                endm
                  .
                  .
                  .
                Problem 2
                  .
                  .
                  .
                Problem Index+2
```

When MASM expands the first invocation of Problem above, it produces the instruction:

```
                mov     ax, Array[2*2]
```

Okay, so far so good. This code loads element two of Array into ax. However, consider the expansion of the second invocation to Problem, above:

```
                mov     ax, Array[Index+2*2]
```

Because MASM's address expressions support operator precedence (see "Operator Precedence" on page 396), this macro expansion will not produce the correct result. It will access the sixth element of Array (at index 12) rather than the tenth element at index 20.

The problem above occurs because MASM simply replaces a formal parameter by the actual parameter's *text*, not the actual parameter's *value*. This *pass by name* parameter passing mechanism should be familiar to long-time C and C++ programmers who use the #define statement. If you think that macro (pass by name) parameters work just like Pascal and C's pass by value parameters, you are setting yourself up for eventual disaster.

One possible solution, that works well for macros like the above, is to put parentheses around macro parameters that occur within expressions inside the macro. Consider the following code:

```
Problem         macro   Parameter
                mov     ax, Array[(Parameter)*2]
                endm
                  .
                  .
                  .
                Problem Index+2
```

This macro invocation expands to

```
                mov     ax, Array[(Index+2)*2]
```

This produces the expected result.

Textual parameter substitution is but one problem you'll run into when using macros. Another problem occurs because MASM has two types of assembly time values: numeric and text. Unfortunately, MASM expects numeric values in some contexts and text values in others. They are not fully interchangeable. Fortunately, MASM provides a set of operators that let you convert between one form and the other (if it is possible to do so). To

understand the subtle differences between these two types of values, look at the following statements:

```
Numeric         =       10+2
Textual         textequ <10+2>
```

MASM evaluates the numeric expression "10+2" and associates the value twelve with the symbol Numeric. For the symbol Textual, MASM simply stores away the string "10+2" and substitutes it for Textual anywhere you use it in an expression.

In many contexts, you could use either symbol. For example, the following two statements both load ax with twelve:

```
                mov     ax, Numeric             ;Same as mov ax, 12
                mov     ax, Textual             ;Same as mov ax, 10+2
```

However, consider the following two statements:

```
                mov     ax, Numeric*2           ;Same as mov ax, 12*2
                mov     ax, Textual*2           ;Same as mov ax, 10+2*2
```

As you can see, the textual substitution that occurs with text equates can lead to the same problems you encountered with textual substitution of macro parameters.

MASM will automatically convert a text object to a numeric value, if the conversion is necessary. Other than the textual substitution problem described above, you can use a text value (whose string represents a numeric quantity) anywhere MASM requires a numeric value.

Going the other direction, numeric value to text value, is not automatic. Therefore, MASM provides an operator you can use to convert numeric data to textual data: the "%" operator. This *expansion* operator forces an immediate evaluation of the following expression and then it converts the result of the expression into a string of digits. Look at these invocations of the Problem macro:

```
                Problem 10+2                    ;Parameter is "10+2"
                Problem %10+2                   ;Parameter is "12"
```

In the second example above, the text expansion operator instructs MASM to evaluate the expression "10+2" and convert the resulting numeric value to a text value consisting of the digits that represent the value twelve. Therefore, these two macro expand into the following statements (respectively):

```
                mov     ax, Array[10+2*2]       ;Problem 10+2 expansion
                mov     ax, Array[12*2]         ;Problem %10+2 expansion
```

MASM provides a second operator, the *substitution* operator that lets you expand macro parameter names where MASM does not normally expect a symbol. The substitution operator is the ampersand ("&") character. If you surround a macro parameter name with ampersands inside a macro, MASM will substitute the parameter's text *regardless of the location of the symbol.* This lets you expand macro parameters whose names appear inside other identifiers or inside literal strings. The following macro demonstrates the use of this operator:

```
DebugMsg        macro   Point, String
Msg&String&     byte    "At point &Point&: &String&"
                endm
                  .
                  .
                  .
                DebugMsg 5, <Assertion fails>
```

The macro invocation immediately above produces the statement:

```
Msg5            byte    "At point 5: Assertion failed"
```

Note how the substitution operator allowed this macro to concatenate "Msg" and "5" to produce the label on the byte directive. Also note that the expansion operator lets you expand macro identifiers even if they appear in a literal string constant. Without the ampersands in the string, MASM would have emitted the statement:

```
Msg5            byte    "At point point: String"
```

Another important operator active within macros is the *literal character operator*, the exclamation mark ("!"). This symbol instructs MASM to pass the following character through without any modification. You would normally use this symbol if you need to include one of the following symbols as a character within a macro:

! & > %

For example, had you really wanted the string in the DebugMsg macro to display the ampersands, you would use the definition:

```
DebugMsg        macro   Point, String
Msg&String&     byte    "At point !&Point!&: !&String!&"
                endm
```

"Debug 5, <Assertion fails>" would produce the following statement:

```
Msg5            byte    "At point &Point&: &String&"
```

Use the "<" and ">" symbols to delimit text data inside MASM. The following two invocations of the PutData macro show how you can use these delimiters in a macro:

```
PutData         macro   TheName, TheData
PD_&TheName&    byte    TheData
                endm
                 .
                 .
                 .
                PutData MyData, 5, 4, 3         ;Emits "PD_MyData byte 5"
                PutData MyData, <5, 4, 3>       ;Emits "PD_MyData byte 5, 4,
3"
```

You can use the text delimiters to surround objects that you wish to treat as a single parameter rather than as a list of multiple parameters. In the PutData example above, the first invocation passes four parameters to PutData (PutData ignores the last two). In the second invocation, there are two parameters, the second consisting of the text 5, 4, 3.

The last macro operator of interest is the ";;" operator. This operator begins a *macro comment*. MASM normally copies all text from the macro into the body of the program during assembly, including all comments. However, if you begin a comment with ";;" rather than a single semicolon, MASM will not expand the comment as part of the code during macro expansion. This increases the speed of assembly by a tiny amount and, more importantly, it does not clutter a program listing with copies of the same comment (see "Controlling the Listing" on page 424 to learn about program listings).

Table 42: Macro Operators

Operator	Description
&	Text substitution operator
< >	Literal text operator
!	Literal character operator
%	Expression operator
;;	Macro comment

8.14.6 A Sample Macro to Implement For Loops

Remember the for loops and matrix operations used in a previous example? At the conclusion of that section there was a brief comment that we could "improve" that code even more using macros, but the example had to wait. With the description of macro operators out of the way, we can now finish that discussion. The macros that implement the for loop are

```
; First, three macros that let us construct symbols by concatenating others.
; This is necessary because this code needs to expand several components in
; text equates multiple times to arrive at the proper symbol.
;
; MakeLbl-      Emits a label create by concatenating the two parameters
;               passed to this macro.

MakeLbl         macro   FirstHalf, SecondHalf
&FirstHalf&&SecondHalf&:
                endm

jgDone          macro   FirstHalf, SecondHalf
                jg      &FirstHalf&&SecondHalf&
                endm

jmpLoop         macro   FirstHalf, SecondHalf
                jmp     &FirstHalf&&SecondHalf&
                endm

; ForLp-        This macro appears at the beginning of the for loop. To invoke
;               this macro, use a statement of the form:
;
;               ForLp   LoopCtrlVar, StartVal, StopVal
;
; Note: "FOR" is a MASM reserved word, which is why this macro doesn't
; use that name.

ForLp           macro   LCV, Start, Stop

; We need to generate a unique, global symbol for each for loop we create.
; This symbol needs to be global because we will need to reference it at the
; bottom of the loop. To generate a unique symbol, this macro concatenates
; "FOR" with the name of the loop control variable and a unique numeric value
; that this macro increments each time the user constructs a for loop with the
; same loop control variable.

                ifndef  $$For&LCV&  ;;Symbol = $$FOR concatenated with LCV
$$For&LCV&      =       0           ;;If this is the first loop w/LCV, use
                else                ;; zero, otherwise increment the value.
$$For&LCV&      =       $$For&LCV& + 1
                endif

; Emit the instructions to initialize the loop control variable:

                mov     ax, Start
                mov     LCV, ax

; Output the label at the top of the for loop. This label takes the form
;               $$FOR LCV x
; where LCV is the name of the loop control variable and X is a unique number
; that this macro increments for each for loop that uses the same loop control
; variable.

                MakeLbl $$For&LCV&, %$$For&LCV&

; Okay, output the code to see if this for loop is complete.
; The jgDone macro generates a jump (if greater) to the label the
; Next macro emits below the bottom of the for loop.

                mov     ax, LCV
                cmp     ax, Stop
                jgDone  $$Next&LCV&, %$$For&LCV&
                endm

; The Next macro terminates the for loop. This macro increments the loop
; control variable and then transfers control back to the label at the top of
; the for loop.

Next            macro   LCV
                inc     LCV
                jmpLoop $$For&LCV&, %$$For&LCV&
                MakeLbl $$Next&LCV&, %$$For&LCV&
                endm
```

With these macros and the LDAX/STAX macros, the code from the array manipulation example presented earlier becomes very simple. It is

```
                ForLp   I, 0, 15
                ForLp   J, 0, 6

                ldax    A, I, J                 ;Fetch A[I][J]
                mov     bx, 15                  ;Compute 16-I.
                sub     bx, I
                ldax    b, bx, J, imul          ;Multiply in B[15-I][J].
                stax    x, J, I                 ;Store to X[J][I]

                Next    J
                Next    I
```

Although this code isn't quite as short as the original C/C++ example, it's getting pretty close!

While the main program became much simpler, there is a question of the macros themselves. The ForLp and Next macros are *extremely* complex! If you had to go through this effort every time you wanted to create a macro, assembly language programs would be ten times harder to write if you decided to use macros. Fortunately, you only have to write (and debug) a macro like this once. Then you can use it as many times as you like, in many different programs, without having to worry much about it's implementation.

Given the complexity of the For and Next macros, it is probably a good idea to *carefully* describe what each statement in these macros is doing. However, before discussing the macros themselves, we should discuss exactly how one might implement a for/next loop in assembly language. This text fully explores the for loop a little later, but we can certainly go over the basics here. Consider the following Pascal for loop:

```
for variable := StartExpression to EndExpression do
        Some_Statement;
```

Pascal begins by computing the value of StartExpression. It then assigns this value to the loop control variable (variable). It then evaluates EndExpression and saves this value in a temporary location. Then the Pascal for statement enters the loop's body. The first thing the loop does is compare the value of variable against the value it computed for EndExpression. If the value of variable is greater than this value for EndExpression, Pascal transfers to the first statement after the for loop, otherwise it executes Some_Statement. After the Pascal for loop executes Some_Statement, it adds one to variable and jumps back to the point where it compares the value of variable against the computed value for EndExpression. Converting this code directly into assembly language yields the following code:

```
;Note: This code assumes StartExpression and EndExpression are simple variables.
;If this is not the case, compute the values for these expression and place
;them in these variables.

                mov     ax, StartExpression
                mov     Variable, ax
ForLoop:        mov     ax, Variable
                cmp     ax, EndExpression
                jg      ForDone

        <Code for Some_Statement>

                inc     Variable
                jmp     ForLoop
ForDone:
```

To implement this as a set of macros, we need to be able to write a short piece of code that will *write* the above assembly language statements for us. At first blush, this would seem easy, why not use the following code?

```
ForLp           macro   Variable, Start, Stop
                mov     ax, Start
                mov     Variable, ax
ForLoop:        mov     ax, Variable
                cmp     ax, Stop
                jg      ForDone
```

```
                endm

Next            macro   Variable
                inc     Variable
                jmp     ForLoop
ForDone:
                endm
```

These two macros would produce correct code – exactly once. However, a problem develops if you try to use these macros a second time. This is particularly evident when using nested loops:

```
                ForLp   I, 1, 10
                ForLp   J, 1, 10
                 .
                 .
                 .
                Next    J
                Next    I
```

The macros above emit the following 80x86 code:

```
                mov     ax, 1       ;The ForLp I, 1, 10
                mov     I, ax       ; macro emits these
ForLoop:        mov     ax, I       ; statements.
                cmp     ax, 10      ;          .
                jg      ForDone     ;          .

                mov     ax, 1       ;The ForLp J, 1, 10
                mov     J, ax       ; macro emits these
ForLoop:        mov     ax, J       ; statements.
                cmp     ax, 10      ;           .
                jg      ForDone     ;           .
                 .
                 .
                 .
                inc     J           ;The Next J macro emits these
                jmp     ForLp       ; statements.
ForDone:
                inc     I           ;The Next I macro emits these
                jmp     ForLp       ; statements.
ForDone:
```

The problem, evident in the code above, is that each time you use the ForLp macro you emit the label "ForLoop" to the code. Likewise, each time you use the Next macro, you emit the label "ForDone" to the code stream. Therefore, if you use these macros more than once (within the same procedure), you will get a duplicate symbol error. To prevent this error, the macros must generate unique labels each time you use them. Unfortunately, the local directive will not work here. The local directive defines a unique symbol *within* a single macro invocation. If you look carefully at the code above, you'll see that the ForLp macro emits a symbol that the code in the Next macro references. Likewise, the Next macro emits a label that the ForLp macro references. Therefore, the label names must be global since the two macros can reference each other's labels.

The solution the actual ForLp and Next macros use is to generate globally known labels of the form "$$For" + "*variable name*" + "*some unique number.*" and "*$$Next*" + "*variable name*" + "*some unique number*". For the example given above, the real ForLp and Next macros would generate the following code:

```
                mov     ax, 1       ;The ForLp I, 1, 10
                mov     I, ax       ; macro emits these
$$ForI0:        mov     ax, I       ; statements.
                cmp     ax, 10      ;          .
                jg      $$NextI0    ;          .

                mov     ax, 1       ;The ForLp J, 1, 10
                mov     J, ax       ; macro emits these
$$ForJ0:        mov     ax, J       ; statements.
                cmp     ax, 10      ;           .
```

```
                jg      $$NextJ0    ;           .
                 .
                 .
                 .
                inc     J           ;The Next J macro emits these
                jmp     $$ForJ0     ; statements.
$$NextJ0:
                inc     I           ;The Next I macro emits these
                jmp     $$ForI0     ; statements.
$$NextI0:
```

The real question is, "How does one generate such labels?"

Constructing a symbol of the form "$$ForI" or "$$NextJ" is pretty easy. Just create a symbol by concatenating the string "$$For" or "$$Next" with the loop control variable's name. The problem occurs when you try to append a numeric value to the end of that string. The actual ForLp and Next code accomplishes this creating assembly time variable names of the form "$$For*variable_name*" and incrementing this variable for each loop with the given loop control variable name. By calling the macros MakeLbl, jgDone, and jmpLoop, ForLp and Next output the appropriate labels and ancillary instructions.

The ForLp and Next macros are very complex. Far more complex than you would typically find in a program. They do, however, demonstrate the power of MASM's macro facilities. By the way, there are *much* better ways to create these symbols using *macro functions*. We'll discuss macro functions next.

8.14.7 Macro Functions

A macro function is a macro whose sole purpose is to return a value for use in the operand field of some other statement. Although there is the obvious parallel between procedures and functions in a high level language and procedural macros and functional macros, the analogy is far from perfect. Macro functions do not let you create sequences of code that emit some instructions that compute a value when the program actually executes. Instead, macro functions simply compute some value at assembly time that MASM can use as an operand.

A good example of a macro function is the Date function. This macro function packs a five bit day, four bit month, and seven bit year value into 16 bits and returns that 16 bit value as the result. If you needed to create an initialized array of dates, you could use code like the following:

```
DateArray       word    Date(2, 4, 84)
                word    Date(1, 1, 94)
                word    Date(7, 20, 60)
                word    Date(7, 19, 69)
                word    Date(6, 18, 74)
                  .
                  .
                  .
```

The Date function would pack the data and the word directive would emit the 16 bit packed value for each date to the object code file. You invoke macro functions by using their name where MASM expects a text expression of some sort. If the macro function requires any parameters, you must enclose them within parentheses, just like the parameters to Date, above.

Macro functions look exactly like standard macros with two exceptions: they do not contain any statements that generate code and they return a text value via an operand to the exitm directive. Note that you *cannot* return a numeric value with a macro function. If you need to return a numeric value, you must first convert it to a text value.

The following macro function implements Date using the 16 bit date format given in Chapter One (see "Bit Fields and Packed Data" on page 28):

```
Date            macro   month, day, year
                local   Value
Value           =       (month shl 12) or (day shl 7) or year
                exitm   %Value
                endm
```

The text expansion operator (“%”) is necessary in the operand field of the exitm directive because macro functions always return textual data, not numeric data. The expansion operator converts the numeric value to a string of digits acceptable to exitm.

One minor problem with the code above is that this function returns garbage if the date isn’t legal. A better design would generate an error if the input date is illegal. You can use the “.err” directive and conditional assembly to do this. The following implementation of Date checks the month, day, and year values to see if they are somewhat reasonable:

```
Date            macro   month, day, year
                local   Value

                if      (month gt 12) or (month lt 1) or \
                        (day gt 31) or (day lt 1) or \
                        (year gt 99) (year lt 1)
                .err
                exitm   <0>         ;;Must return something!
                endif

Value           =       (month shl 12) or (day shl 7) or year
                exitm   %Value
                endm
```

With this version, any attempt to specify a totally outrageous date triggers the assembly of the “.err” directive that forces an error at assembly time.

8.14.8 Predefined Macros, Macro Functions, and Symbols

MASM provides four built-in macros and four corresponding macro functions. In addition, MASM also provides a large number of predefined symbols you can access during assembly. Although you would rarely use these macros, functions, and variables outside of moderately complex macros, they are essential when you do need them.

Table 43: MASM Predefined Macros

Name	operands	Example	Description
substr	string, start, length Returns: text data	NewStr substr Oldstr, 1, 3	Returns a string consisting of the characters from start to start+length in the string operand. The length operand is optional. If it is not present, MASM returns all characters from position start through the end of the string.
instr	start, string, substr Returns: numeric data	Pos instr 2, OldStr, <ax>	Searches for “substr” within “string” starting at position “start.” The starting value is optional. If it is missing, MASM begins searching for the string from position one. If MASM cannot find the substring within the string operand, it returns the value zero.
sizestr	string Returns: numeric data	StrSize sizestr OldStr	Returns the size of the string in the operand field.
catstr	string, string, ... Returns: text data	NewStr catstr OldStr, <$$>	Creates a new string by concatenating each of the strings appearing in the operand field of the catstr macro.

The substr and catstr macros return text data. In some respects, they are similar to the textequ directive since you use them to assign textual data to a symbol at assembly time. The instr and sizestr are similar to the "=" directive insofar as they return a numeric value.

The catstr macro can eliminate the need for the MakeLbl macro found in the ForLp macro. Compare the following version of ForLp to the previous version (see "A Sample Macro to Implement For Loops" on page 409).

```
ForLp           macro   LCV, Start, Stop
                local   ForLoop

                ifndef  $$For&LCV&
$$For&LCV&      =       0
                else
$$For&LCV&      =       $$For&LCV& + 1
                endif

                mov     ax, Start
                mov     LCV, ax

; Due to bug in MASM, this won't actually work. The idea is sound, though
; Read on for correct solution.

ForLoop         textequ @catstr($For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov     ax, LCV
                cmp     ax, Stop
                jgDone  $$Next&LCV&, %$$For&LCV&
                endm
```

MASM also provides macro function forms for catstr, instr, sizestr, and substr. To differentiate these macro functions from the corresponding predefined macros, MASM uses the names @catstr, @instr, @sizestr, and @substr. The the following equivalences between these operations:

```
Symbol          catstr  String1, String2, ...
Symbol          textequ @catstr(String1, String2, ...)

Symbol          substr  SomeStr, 1, 5
Symbol          textequ @substr(SomeStr, 1, 5)

Symbol          instr   1, SomeStr, SearchStr
Symbol          =       @substr(1, SomeStr, SearchStr)

Symbol          sizestr SomeStr
Symbol          =       @sizestr(SomeStr)
```

Table 44: MASM Predefined Macro Functions

Name	Parameters	Example
@substr	string, start, length Returns: text data	ifidn @substr(parm, 1, 4), <[bx]>
@instr	start, string, substr Returns: numeric data	if @instr(parm,<bx>)
@sizestr	string Returns: numeric data	byte @sizestr(SomeStr)
@catstr	string, string, ... Returns: text data	jg @catstr($$Next&LCV&, %$$For&LCV&)

The last example above shows how to get rid of the jgDone and jmpLoop macros in the ForLp macro. A final, improved, version of the ForLp and Next macros, eliminating the three support macros and working around the bug in MASM might look something like the following:

```
ForLp           macro   LCV, Start, Stop
                local   ForLoop

                ifndef  $$For&LCV&
$$For&LCV&      =       0
                else
$$For&LCV&      =       $$For&LCV& + 1
                endif

                mov     ax, Start
                mov     LCV, ax

ForLoop         textequ @catstr($For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov     ax, LCV
                cmp     ax, Stop
                jg      @catstr($$Next&LCV&, %$$For&LCV&)
                endm

Next            macro   LCV
                local   NextLbl
                inc     LCV
                jmp     @catstr($$For&LCV&, %$$For&LCV&)
NextLbl         texteq  @catstr($Next&LCV&, %$$For&LCV&)
&NextLbl&:
                endm
```

MASM also provides a large number of built in variables that return information about the current assembly. The following table describes these built in assembly time variables.

Table 45: MASM Predefined Assembly Time Variables

Category	Name	Description	Return result
Date & Time Information	@Date	Returns the date of assembly.	Text value
	@Time	Returns a string denoting the time of assembly.	Text value

Table 45: MASM Predefined Assembly Time Variables

Category	Name	Description	Return result
Environment Information	@CPU	Returns a 16 bit value whose bits determine the active processor directive. Specifying the .8086, .186, .286, .386, .486, and .586 directives enable additional instructions in MASM. They also set the corresponding bits in the @cpu variable. Note that MASM sets *all* the bits for the processors it can handle at any one given time. For example, if you use the .386 directive, MASM sets bits zero, one, two, and three in the @cpu variable.	Bit 0 - 8086 instrs permissible. Bit 1 - 80186 instrs permissible. Bit 2 - 80286 instrs permissible. Bit 3- 80386 instrs permissible. Bit 4- 80486 instrs permissible. Bit 5- Pentium instrs permissible. Bit 6- Reserved for 80686 (?). Bit 7- Protected mode instrs okay. Bit 8- 8087 instrs permissible. Bit 10- 80287 instrs permissible. Bit 11- 80386 instrs permissible. (bit 11 is also set for 80486 and Pentium instr sets).
	@Environ	@Environ(name) returns the text associated with DOS environment variable name. The parameter must be a text value that evaluates to a valid DOS environment variable name.	Text value
	@Interface	Returns a numeric value denoting the current language type in use. Note that this information is similar to that provided by the opattr attribute. The H.O. bit determines if you are assembling code for MS-DOS/Windows or OS/2. This directive is mainly useful for those using MASM's simplified segment directives. Since this text does not deal with the simplified directives, further discussion of this variable is unwarranted.	Bits 0-2 000- No language type 001- C 010- SYSCALL 011- STDCALL 100- Pascal 101- FORTRAN 110- BASIC Bit 7 0- MS-DOS or Windows 1- OS/2
	@Version	Returns a numeric value that is the current MASM version number multiplied by 100. For example, MASM 6.11's @version variable returns 611.	Numeric value
File Information	@FileCur	Returns the current source or include file name, including any necessary pathname information.	Text value
	@File-Name	Returns the current source file name (base name only, no path information). If in an include file, this variable returns the name of the source file that included the current file.	Text value
	@Line	Returns the current line number in the source file.	Numeric value

Table 45: MASM Predefined Assembly Time Variables

Category	Name	Description	Return result
Segment [a] Information	@code	Returns the name of the current code segment.	Text value
	@data	Returns the name of the current data segment.	Text value
	@FarData?	Returns the name of the current far data segment.	Text value
	@Word-Size	Returns two if this is a 16 bit segment, four if this is a 32 bit segment.	Numeric value
	@Code-Size	Returns zero for Tiny, Small, Compact, and Flat models. Returns one for Medium, Large, and Huge models.	Numeric value
	@DataSize	Returns zero for Tiny, Small, Medium, and Flat memory models. Returns one for Compact and Large models. Returns two for Huge model programs.	Numeric value
	@Model	Returns one for Tiny model, two for Small model, three for Compact model, four for Medium model, five for Large model, six for Huge model, and seven for Flag model.	Numeric value
	@CurSeg	Returns the name of the current code segment.	Text value
	@stack	The name of the current stack segment.	Text value

a. These functions are intended for use with MASM's simplified segment directives. This chapter does not discuss these directives, so these functions will probably be of little use.

Although there is insufficient space to go into detail about the possible uses for each of these variables, a few examples might demonstrate some of the possibilities. Other uses of these variables will appear throughout the text; however, the most impressive uses will be the ones *you* discover.

The @CPU variable is quite useful if you want to assemble different code sequences in your program for different processors. The section on conditional assembly in this chapter described how you could create a symbol to determine if you are assembling the code for an 80386 and later processor or a stock 8086 processor. The @CPU symbol provides a symbol that will tell you *exactly* which instructions are allowable at any given point in your program. The following is a rework of that example using the @CPU variable:

```
if      @CPU and 100b ;Need an 80286 or later processor
shl     ax, 4         ; for this instruction.
else                  ;Must be 8086 processor.
mov     cl, 4
shl     ax, cl
endif
```

You can use the @Line directive to put special diagnostic messages in your code. The following code would print an error message including the line number in the source file of the offending assertion, if it detects an error at run-time:

```
mov     ax, ErrorFlag
cmp     ax, 0
je      NoError
mov     ax, @Line     ;Load AX with current line #
call    PrintError    ;Go print error message and Line #
jmp     Quit          ;Terminate program.
```

8.14.9 Macros vs. Text Equates

Macros, macro functions, and text equates all substitute text in a program. While there is some overlap between them, they really do serve different purposes in an assembly language program.

Text equates perform a single text substitution on a line. They do not allow any parameters. However, you can replace text *anywhere* on a line with a text equate. You can expand a text equate in the label, mnemonic, operand, or even the comment field. Furthermore, you can replace multiple fields, even an entire line with a single symbol.

Macro functions are legal in the operand field only. However, you can pass parameters to macro functions making them considerably more general than simple text equates.

Procedural macros let you emit sequences of statements (with text equates you can emit, at most, one statement).

8.14.10 Macros: Good and Bad News

Macros offer considerable convenience. They let you insert several instructions into your source file by simply typing a single command. This can save you an incredible amount of typing when entering huge tables, each line of which contains some bizarre, but repeated calculation. It's useful (in certain cases) for helping make your programs more readable. Few would argue that ForLp I,1,10 is not more readable than the corresponding 80x86 code. Unfortunately, it's easy to get carried away and produce code that is inefficient, hard to read, and hard to maintain.

A lot of so-called "advanced" assembly language programmers get carried away with the idea that they can create their own instructions via macro definitions and they start creating macros for every imaginable function under the sun. The COPY macro presented earlier is a good example. The 80x86 doesn't support a memory to memory move operation. Fine, we'll create a macro that does the job for us. Soon, the assembly language program doesn't look like 80x86 assembly language at all. Instead, a large number of the statements are macro invocations. Now this may be great for the programmer who has created all these macros and intimately understands their operation. To the 80x86 programmer who isn't familiar with those macros, however, it's all gibberish. Maintaining a program someone else wrote, that contains "new" instructions implemented via macros, is a horrible task. Therefore, you should rarely use macros as a device to create new instructions on the 80x86.

Another problem with macros is that they tend to hide side effects. Consider the COPY macro presented earlier. If you encountered a statement of the form COPY VAR1,VAR2 in an assembly language program, you'd think that this was an innocuous statement that copies VAR2 to VAR1. Wrong! It also destroys the current contents of the ax register leaving a copy of the value in VAR2 in the ax register. This macro invocation doesn't make this very clear. Consider the following code sequence:

```
mov         ax, 5
copy        Var2, Var1
mov         Var1, ax
```

This code sequence copies Var1 into Var2 and then (supposedly) stores five into Var1. Unfortunately, the COPY macro has wiped out the value in ax (leaving the value originally contained in Var1 alone), so this instruction sequence does not modify Var1 at all!

Another problem with macros is efficiency. Consider the following invocations of the COPY macro:

```
copy        Var3, Var1
copy        Var2, Var1
copy        Var0, Var1
```

These three statements generate the code:

```
mov         ax, Var1
mov         Var3, ax
mov         ax, Var1
mov         Var2, ax
mov         ax, Var1
mov         Var0, ax
```

Clearly, the last two mov ax,Var1 instructions are superfluous. The ax register already contains a copy of Var1, there is no need to reload ax with this value. Unfortunately, this inefficiency, while perfectly obvious in the expanded code, isn't obvious at all in the macro invocations.

Another problem with macros is complexity. In order to generate efficient code, you can create extremely complex macros using conditional assembly (especially ifb, ifidn, etc.), repeat loops (described a little later), and other directives. Unfortunately, these macros are small programs all on their own. You can have bugs in your macros just as you can have bugs in your assembly language program. And the more complex your macros become, the more likely they'll contain bugs that will, of course, become bugs in your program when invoking the macro.

Overusing macros, especially complex ones, produces hard to read code that is hard to maintain. Despite the enthusiastic claims of those who love macros, the unbridled use of macros within a program generally causes more bugs than it helps to prevent. If you're going to use macros, go easy on them.

There is a good side to macros, however. If you standardize on a set of macros and document all your programs as using these macros, they may help make your programs more readable. Especially if those macros have easily identifiable names. The *UCR Standard Library for 80x86 Assembly Language Programmers* uses macros for most library calls. You'll read more about the UCR Standard Library in the next chapter.

8.15 Repeat Operations

Another macro format (at least by Microsoft's definition) is the repeat macro. A repeat macro is nothing more than a loop that repeats the statements within the loop some specified number of times. There are three types of repeat macros provided by MASM: repeat/rept, for/irp, and forc/irpc. The repeat/rept macro uses the following syntax:

```
        repeat      expression
<statements>
        endm
```

Expression must be a numeric expression that evaluates to an unsigned constant. The repeat directive duplicates all the statements between repeat and endm that many times. The following code generates a table of 26 bytes containing the 26 uppercase characters:

```
ASCIICode       =           'A'
                repeat      26
                byte        ASCIICode
ASCIICode       =           ASCIICode+1
                endm
```

The symbol ASCIICode is assigned the ASCII code for "A". The loop repeats 26 times, each time emitting a byte with the value of ASCIICode. Also, the loop increments the ASCIICode symbol on each repetition so that it contains the ASCII code of the next character in the ASCII table. This effectively generates the following statements:

```
                byte        'A'
                byte        'B'
                  .
                  .
                  .
                byte        'Y'
                byte        'Z'
ASCIICode       =           27
```

Note that the repeat loop executes at assembly time, not at run time. Repeat is not a mechanism for creating loops within your program; use it for replicating sections of code within your program. If you want to create a loop that executes some number of times within your program, use the loop instruction. Although the following two code sequences produce the same result, they are *not* the same:

```
; Code sequence using a run-time loop:

                mov     cx, 10
AddLp:          add     ax, [bx]
                add     bx, 2
                loop    AddLp

; Code sequence using an assembly-time loop:

                repeat  10
                add     ax, [bx]
                add     bx, 2
                endm
```

The first code sequence above emits four machine instructions to the object code file. At assembly time, the 80x86 CPU executes the statements between AddLp and the loop instruction ten times under the control of the loop instruction. The second code sequence above emits 20 instructions to the object code file. At run time, the 80x86 CPU simply executes these 20 instructions sequentially, with no control transfer. The second form will be faster, since the 80x86 does not have to execute the loop instruction every third instruction. On the other hand, the second version is also much larger because it replicates the body of the loop ten times in the object code file.

Unlike standard macros, you do not define and invoke repeat macros separately. MASM emits the code between the repeat and endm directives upon encountering the repeat directive. There isn't a separate invocation phase. If you want to create a repeat macro that can be invoked throughout your program, consider the following:

```
REPTMacro       macro       Count
                repeat      Count
             <statements>
                endm
                endm
```

By placing the repeat macro inside a standard macro, you can invoke the repeat macro anywhere in your program by invoking the REPTMacro macro. Note that you need two endm directives, one to terminate the repeat macro, one to terminate the standard macro.

Rept is a synonym for repeat. Repeat is the newer form, MASM supports Rept for compatibility with older source files. You should always use the repeat form.

8.16 The FOR and FORC Macro Operations

Another form of the repeat macro is the for macro. This macro takes the following form:

```
                for     parameter,<item1 {,item2 {,item3 {,...}}}>
            <statements>
                endm
```

The angle brackets are required around the items in the operand field of the for directive. The braces surround optional items, the braces should not appear in the operand field.

The for directive replicates the instructions between for and endm once for each item appearing in the operand field. Furthermore, for each iteration, the first symbol in the operand field is assigned the value of the successive items from the second parameter. Consider the following loop:

```
                for     value,<0,1,2,3,4,5>
                byte    value
                endm
```

This loop emits six bytes containing the values zero, one, two, ..., five. It is absolutely identical to the sequence of instructions:

```
byte       0
byte       1
byte       2
byte       3
byte       4
byte       5
```

Remember, the for loop, like the repeat loop, executes at assembly time, not at run time.

For's second operand need not be a literal text constant; you can supply a macro parameter, macro function result, or a text equate for this value. Keep in mind, though, that this parameter *must* expand to a text value with the text delimiters around it.

Irp is an older, obsolete, synonym for for. MASM allows irp to provide compatibility with older source code. However, you should always use the for directive.

The third form of the loop macro is the forc macro. It differs from the for macro in that it repeats a loop the number of times specified by the length of a character string rather than by the number of operands present. The syntax for the forc directive is

```
forc     parameter,<string>

<statements>

endm
```

The statements in the loop repeat once for each character in the string operand. The angle brackets must appear around the string. Consider the following loop:

```
forc     value,<012345>
byte       value
endm
```

This loop produces the same code as the example for the for directive above.

Irpc is an old synonym for forc provided for compatibility reasons. You should always use forc in your new code.

8.17 The WHILE Macro Operation

The while macro lets you repeat a sequence of code in your assembly language file an indefinite number of times. An assembly time expression, that while evaluates before emitting the code for each loop, determines whether it repeats. The syntax for this macro is

```
    while    expression
<Statements>
    endm
```

This macro evaluates the assembly-time expression; if this expression's value is zero, the while macro ignores the statements up to the corresponding endm directive. If the expression evaluates to a non-zero value (true), then MASM assembles the statements up to the endm directive and reevaluates the expression to see if it should assemble the body of the while loop again.

Normally, the while directive repeats the statements between the while and endm as long as the expression evaluates true. However, you can also use the exitm directive to prematurely terminate the expansion of the loop body. Keep in mind that you need to provide *some* condition that terminates the loop, otherwise MASM will go into an infinite loop and continually emit code to the object code file until the disk fills up (or it will simply go into an infinite loop if the loop does not emit any code).

8.18 Macro Parameters

Standard MASM macros are very flexible. If the number of actual parameters (those supplied in the operand field of the macro invocation) does not match the number of for-

mal parameters (those appearing in the operand field of the macro definition), MASM won't necessarily complain. If there are more actual parameters than formal parameters, MASM ignores the extra parameters and generates a warning. If there are more formal parameters than actual parameters, MASM substitutes the empty string ("<>") for the extra formal parameters. By using the ifb and ifnb conditional assembly directives, you can test this last condition. While this parameter substitution technique is flexible, it also leaves open the possibility of error. If you want to require that the programmer supply exactly three parameters and they actually supply less, MASM will not generate an error. If you forget to test for the presence of each parameter using ifb, you could generate bad code. To overcome this limitation, MASM provides the ability to specify that certain macro parameters are required. You can also assign a default value to a parameter if the programming doesn't supply one. Finally, MASM also provides facilities to allow a variable number of macro arguments.

If you want to require a programmer to supply a particular macro parameters, simply put ":req" after the macro parameter in the macro definition. At assembly time, MASM will generate an error if that particular macro is missing.

```
Needs2Parms     macro   parm1:req, parm2:req
                  .
                  .
                  .
                endm
                  .
                  .
                  .
                Needs2Parms ax                  ;Generates an error.
                Needs2Parms                     ;Generates an error.
                Needs2Parms ax, bx              ;Works fine.
```

Another possibility is to have the macro supply a default value for a macro if it is missing from the actual parameter list. To do this, simply use the ":=<*text*>" operator immediately after the parameter name in the formal parameter list. For example, the int 10h BIOS function provides various video services. One of the most commonly used video services is the ah=0eh function that outputs the character in al to the video display. The following macro lets the caller specify which function they want to use, and defaults to function 0eh if they don't specify a parameter:

```
Video           macro   service := <0eh>
                mov     ah, service
                int     10h
                endm
```

The last feature MASM's macros support is the ability to process a variable number of parameters. To do this you simply place the operator ":vararg" after the *last* formal parameter in the parameter list. MASM associates the first *n* actual parameters with the corresponding formal parameters appearing before the variable argument, it then creates a text equate of all remaining parameters to the formal parameter suffixed with the ":vararg" operator. You can use the for macro to extract each parameter from this variable argument list. For example, the following macro lets you declare an arbitrary number of two dimensional arrays, all the same size. The first two parameters specify the number of rows and columns, the remaining optional parameters specify the names of the arrays:

```
MkArrays        macro   NumRows:req, NumCols:req, Names:vararg
                for     AryName, Names
&AryName&       word    NumRows dup (NumCols dup (?))
                endm
                endm

                  .
                  .
                  .
                MkArrays 8, 12, A, B, X, Y
```

8.19 Controlling the Listing

MASM provides several assembler directives that are useful for controlling the output of the assembler. These directives include echo, %out, title, subttl, page, .list, .nolist, and .xlist. There are several others, but these are the most important.

8.19.1 The ECHO and %OUT Directives

The echo and %out directives simply print whatever appears in its operand field to the video display during assembly. Some examples of echo and %out appeared in the sections on conditional assembly and macros. Note that %out is an older form of echo provided for compatibility with old source code.. You should use echo in all your new code.

8.19.2 The TITLE Directive

The title assembler directive assigns a title to your source file. Only one title directive may appear in your program. The syntax for this directive is

```
title       text
```

MASM will print the specified text at the top of each page of the assembled listing.

8.19.3 The SUBTTL Directive

The subttl (subtitle) directive is similar to the title directive, except multiple subtitles may appear within your source file. Subtitles appear immediately below the title at the top of each page in the assembled listing. The syntax for the subttl directive is

```
subttl      text
```

The specified text will become the new subtitle. Note that MASM will not print the new subtitle until the next page eject. If you wish to place the subtitle on the same page as the code immediately following the directive, use the page directive (described next) to force a page ejection.

8.19.4 The PAGE Directive

The page directive performs two functions- it can force a page eject in the assembly listing and it can set the width and length of the output device. To force a page eject, the following form of the page directive is used:

```
page
```

If you place a plus sign, "+", in the operand field, then MASM performs a page break, increments the section number, and resets the page number to one. MASM prints page numbers using the format

section-page

If you want to take advantage of the section number facility, you will have to manually insert page breaks (with a "+" operand) in front of each new section.

The second form of the page command lets you set the printer page width and length values. It takes the form:

```
page      length, width
```

where length is the number of lines per page (defaults to 50, but 56-60 is a better choice for most printers) and width is the number of characters per line. The default page width is

80 characters. If your printer is capable of printing 132 columns, you should change this value to 132 so your listings will be easier to read. Note that some printers, even if their carriage is only 8-1/2" wide, will print at least 132 columns across in a condensed mode. Typically some control character must be sent to the printer to place it in condensed mode. You can insert such a control character in a comment at the beginning of your source listing.

8.19.5 The .LIST, .NOLIST, and .XLIST Directives

The .list, .nolist, and .xlist directives can be used to selectively list portions of your source file during assembly. .List turns the listing on, .Nolist turns the listing off. .Xlist is an obsolete form of .Nolist for older code.

By sprinkling these three directives throughout your source file, you can list only those sections of code that interest you. None of these directives accept any operands. They take the following forms:

```
.list
.nolist
.xlist
```

8.19.6 Other Listing Directives

MASM provides several other listing control directives that this chapter will not cover. These let you control the output of macros, conditional assembly segments, and so on to the listing file. Please see the appendices for details on these directives.

8.20 Managing Large Programs

Most assembly language programs are not totally stand alone programs. In general, you will call various standard library or other routines which are not defined in your main program. For example, you've probably noticed by now that the 80x86 doesn't provide any instructions like "read", "write", or "printf" for doing I/O operations. In fact, the only instructions you've seen that do I/O include the 80x86 in and out instructions, which are really just special mov instructions, and the echo/%out directives that perform assembly-time output, not the run-time output you want. Is there no way to do I/O from assembly language? Of course there is. You can write procedures that perform the I/O operations like "read" and "write". Unfortunately, writing such routines is a complex task, and beginning assembly language programmers are not ready for such tasks. That's where the UCR Standard Library for 80x86 Assembly Language Programmers comes in. This is a package of procedures you can call to perform simple I/O operations like "printf".

The UCR Standard Library contains thousands of lines of source code. Imagine how difficult programming would be if you had to merge these thousands of lines of code into your simple programs. Fortunately, you don't have to.

For small programs, working with a single source file is fine. For large programs this gets very cumbersome (consider the example above of having to include the entire UCR Standard Library into each of your programs). Furthermore, once you've debugged and tested a large section of your code, continuing to assemble that same code when you make a small change to some other part of your program is a waste of time. The UCR Standard Library, for example, takes several minutes to assemble, even on a fast machine. Imagine having to wait five or ten minutes on a fast Pentium machine to assemble a program to which you've made a one line change!

As with HLLs, the solution is *separate compilation* (or *separate assembly* in MASM's case). First, you break up your large source files into manageable chunks. Then you

assemble the separate files into object code modules. Finally, you link the object modules together to form a complete program. If you need to make a small change to one of the modules, you only need to reassemble that one module, you do not need to reassemble the entire program.

The UCR Standard Library works in precisely this way. The Standard Library is already assembled and ready to use. You simply call routines in the Standard Library and link your code with the Standard Library using a *linker* program. This saves a tremendous amount of time when developing a program that uses the Standard Library code. Of course, you can easily create your own object modules and link them together with your code. You could even add new routines to the Standard Library so they will be available for use in future programs you write.

"Programming in the large" is a term software engineers have coined to describe the processes, methodologies, and tools for handling the development of large software projects. While everyone has their own idea of what "large" is, separate compilation, and some conventions for using separate compilation, are one of the big techniques for "programming in the large." The following sections describe the tools MASM provides for separate compilation and how to effectively employ these tools in your programs.

8.20.1 The INCLUDE Directive

The include directive, when encountered in a source file, switches program input from the current file to the file specified in the parameter list of the include. This allows you to construct text files containing common equates, macros, source code, and other assembler items, and include such a file into the assembly of several separate programs. The syntax for the include directive is

```
include     filename
```

Filename must be a valid DOS filename. MASM merges the specified file into the assembly at the point of the include directive. Note that you can nest include statements inside files you include. That is, a file being included into another file during assembly may itself include a third file.

Using the include directive by itself does not provide separate compilation. You *could* use the include directive to break up a large source file into separate modules and join these modules together when you assemble your file. The following example would include the PRINTF.ASM and PUTC.ASM files during the assembly of your program:

```
            include     printf.asm
            include     putc.asm

    <Code for your program goes here>

            end
```

Now your program *will* benefit from the modularity gained by this approach. Alas, you will not save any development time. The include directive inserts the source file at the point of the include during assembly, exactly as though you had typed that code in yourself. MASM still has to assemble the code and that takes time. Were you to include all the files for the Standard Library routines, your assemblies would take *forever*.

In general, you should *not* use the include directive to include source code as shown above[16]. Instead, you should use the include directive to insert a common set of constants (equates), macros, external procedure declarations, and other such items into a program. Typically an assembly language include file does *not* contain any machine code (outside of a macro). The purpose of using include files in this manner will become clearer after you see how the public and external declarations work.

16. There is nothing wrong with this, other than the fact that it does not take advantage of separate compilation.

8.20.2 The PUBLIC, EXTERN, and EXTRN Directives

Technically, the include directive provides you with all the facilities you need to create modular programs. You can build up a library of modules, each containing some specific routine, and include any necessary modules into an assembly language program using the appropriate include commands. MASM (and the accompanying LINK program) provides a better way: external and public symbols.

One major problem with the include mechanism is that once you've debugged a routine, including it into an assembly wastes a lot of time since MASM must reassemble bug-free code every time you assemble the main program. A much better solution would be to preassemble the debugged modules and link the object code modules together rather than reassembling the entire program every time you change a single module. This is what the public and extern directives provide for you. Extrn is an older directive that is a synonym for extern. It provides compatibility with old source files. You should always use the extern directive in new source code.

To use the public and extern facilities, you must create at least two source files. One file contains a set of variables and procedures used by the second. The second file uses those variables and procedures without knowing how they're implemented. To demonstrate, consider the following two modules:

```
;Module #1:

                public      Var1, Var2, Proc1
DSEG            segment     para public 'data'
Var1            word        ?
Var2            word        ?
DSEG            ends

CSEG            segment     para public 'code'
                assume      cs:cseg, ds:dseg
Proc1           proc        near
                mov         ax, Var1
                add         ax, Var2
                mov         Var1, ax
                ret
Proc1           endp
CSEG            ends
                end

;Module #2:
                extern      Var1:word, Var2:word, Proc1:near
CSEG            segment     para public 'code'
                  .
                  .
                  .
                mov         Var1, 2
                mov         Var2, 3
                call        Proc1
                  .
                  .
                  .
CSEG            ends
                end
```

Module #2 references Var1, Var2, and Proc1, yet these symbols are external to module #2. Therefore, you must declare them external with the extern directive. This directive takes the following form:

```
                extern      name:type {,name:type...}
```

Name is the name of the external symbol, and type is the type of that symbol. Type may be any of near, far, proc, byte, word, dword, qword, tbyte, abs (absolute, which is a constant), or some other user defined type.

The current module uses this type declaration. Neither MASM nor the linker checks the declared type against the module defining name to see if the types agree. Therefore, you must exercise caution when defining external symbols. The public directive lets you export a symbol's value to external modules. A public declaration takes the form:

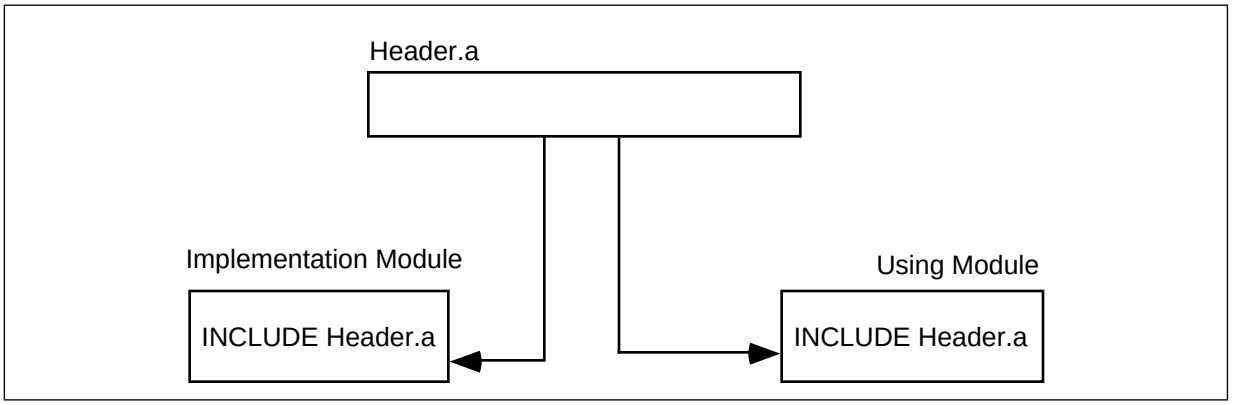


Figure 8.8 Using a Single Include file for Implementation and Using Modules

```
public      name {,name ...}
```

Each symbol appearing in the operand field of the public statement is available as an external symbol to another module. Likewise, all external symbols within a module must appear within a public statement in some other module.

Once you create the source modules, you should assemble the file containing the public declarations first. With MASM 6.x, you would use a command like

ML /c pubs.asm

The "/c" option tells MASM to perform a "compile-only" assembly. That is, it will not try to link the code after a successful assembly. This produces a "pubs.obj" object module.

Next, assemble the file containing the external definitions and link in the code using the MASM command:

ML exts.asm pubs.obj

Assuming there are no errors, this will produce a file "exts.exe" which is the linked and executable form of the program.

Note that the extern directive defines a symbol in your source file. Any attempt to redefine that symbol elsewhere in your program will produce a "duplicate symbol" error. This, as it turns out, is the source of problems which Microsoft solved with the externdef directive.

8.20.3 The EXTERNDEF Directive

The externdef directive is a combination of public and extern all rolled into one. It uses the same syntax as the extern directive, that is, you place a list of name:type entries in the operand field. If MASM does not encounter another definition of the symbol in the current source file, externdef behaves exactly like the extern statement. If the symbol does appear in the source file, then externdef behaves like the public command. With externdef there really is no need to use the public or extern statements unless you feel somehow compelled to do so.

The important benefit of the externdef directive is that it lets you minimize duplication of effort in your source files. Suppose, for example, you want to create a module with a bunch of support routines for other programs. In addition to sharing some routines and some variables, suppose you want to share constants and macros as well. The include file mechanism provides a perfect way to handle this. You simply create an include file containing the constants, macros, and externdef definitions and include this file in the module that implements your routines and in the modules that use those routines (see Figure 8.8).

Note that extern and public wouldn't work in this case because the implementation module needs the public directive and the using module needs the extern directive. You would have to create two separate header files. Maintaining two separate header files that

contain mostly identical definitions is not a good idea. The externdef directive provides a solution.

Within your headers files you should create segment definitions that match those in the including modules. Be sure to put the externdef directives inside the same segments in which the symbol is actually defined. This associates a segment value with the symbol so that MASM can properly make appropriate optimizations and other calculations based on the symbol's full address:

```
; From "HEADER.A" file:

cseg            segment     para public 'code'
                externdef   Routine1:near, Routine2:far
cseg            ends
dseg            segment     para public 'data'
                externdef   i:word, b:byte, flag:byte
dseg            ends
```

This text adopts the UCR Standard Library convention of using an ".a" suffix for assembly language header files. Other common suffixes in use include ".inc" and ".def".

8.21 Make Files

Although using separate compilation reduces assembly time and promotes code reuse and modularity, it is not without its own drawbacks. Suppose you have a program that consists of two modules: pgma.asm and pgmb.asm. Also suppose that you've already assembled both modules so that the files pgma.obj and pgmb.obj exist. Finally, you make changes to pgma.asm and pgmb.asm and assemble the pgma.asm *but forget to assemble the pgmb.asm file*. Therefore, the pgmb.obj file will be *out of date* since this object file does not reflect the changes made to the pgmb.asm file. If you link the program's modules together, the resulting .exe file will only contain the changes to the pgma.asm file, it will not have the updated object code associated with pgmb.asm. As projects get larger, as they have more modules associated with them, and as more programmers begin working on the project, it gets very difficult to keep track of which object modules are up to date.

This complexity would normally cause someone to reassemble (or recompile) *all* modules in a project, even if many of the .obj files are up to date, simply because it might seem too difficult to keep track of which modules are up to date and which are not. Doing so, of course, would eliminate many of the benefits that separate compilation offers. Fortunately, there is a tool that can help you manage large projects: nmake. The nmake program, will a little help from you, can figure out which files need to be reassemble and which files have up to date .obj files. With a properly defined *make file*, you can easily assemble only those modules that absolutely must be assembled to generate a consistent program.

A make file is a text file that lists assembly-time dependencies between files. An .exe file, for example, is *dependent* on the source code whose assembly produce the executable. If you make any changes to the source code you will (probably) need to reassemble or recompile the source code to produce a new .exe file[17].

Typical dependencies include the following:

- An executable file (.exe) generally depends only on the set of object files (.obj) that the linker combines to form the executable.
- A given object code file (.obj) depends on the assembly language source files that were assembled to produce that object file. This includes the

17. Obviously, if you only change comments or other statements in the source file that do not affect the executable file, a recompile or reassembly will not be necessary. To be safe, though, we will assume *any* change to the source file will require a reassembly.

assembly language source files (.asm) and any files included during that assembly (generally .a files).

- The source files and include files generally don't depend on anything.

A make file generally consists of a dependency statement followed by a set of commands to handle that dependency. A dependency statement takes the following form:

dependent-file : *list of files*

Example:

```
pgm.exe: pgma.obj pgmb.obj
```

This statement says that "pgm.exe" is dependent upon pgma.obj and pgmb.obj. Any changes that occur to pgma.obj or pgmb.obj will require the generate of a new pgm.exe file.

The nmake.exe program uses a *time/date stamp* to determine if a dependent file is out of date with respect to the files it depends upon. Any time you make a change to a file, MS-DOS and Windows will update a *modification time and date* associated with the file. The nmake.exe program compares the modification date/time stamp of the dependent file against the modification date/time stamp of the files it depends upon. If the dependent file's modification date/time is earlier than one or more of the files it depends upon, or one of the files it depends upon is not present, then nmake.exe assumes that some operation must be necessary to update the dependent file.

When an update is necessary, nmake.exe executes the set of (MS-DOS) commands following the dependency statement. Presumably, these commands would do whatever is necessary to produce the updated file.

The dependency statement *must* begin in column one. Any commands that must execute to resolve the dependency must start on the line immediately following the dependency statement and each command must be indented one tabstop. The pgm.exe statement above would probably look something like the following:

```
pgm.exe: pgma.obj pgmb.obj
	ml /Fepgm.exe pgma.obj pgmb.obj
```

(The "/Fepgm.exe" option tells MASM to name the executable file "pgm.exe.")

If you need to execute more than one command to resolve the dependencies, you can place several commands after the dependency statement in the appropriate order. Note that you must indent all commands one tab stop. Nmake.exe ignores any blank lines in a make file. Therefore, you can add blank lines, as appropriate, to make the file easier to read and understand.

There can be more than a single dependency statement in a make file. In the example above, for example, pgm.exe depends upon the pgma.obj and pgmb.obj files. Obviously, the .obj files depend upon the source files that generated them. Therefore, before attempting to resolve the dependencies for pgm.exe, nmake.exe will first check out the rest of the make file to see if pgma.obj or pgmb.obj depends on anything. If they do, nmake.exe will resolve those dependencies first. Consider the following make file:

```
pgm.exe: pgma.obj pgmb.obj
	ml /Fepgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.asm
	ml /c pgma.asm

pgmb.obj: pgmb.asm
	ml /c pgmb.asm
```

The nmake.exe program will process the first dependency line it finds in the file. However, the files pgm.exe depends upon themselves have dependency lines. Therefore, nmake.exe will first ensure that pgma.obj and pgmb.obj are up to date before attempting to execute MASM to link these files together. Therefore, if the only change you've made has been to pgmb.asm, nmake.exe takes the following steps (assuming pgma.obj exists and is up to date).

1. Nmake.exe processes the first dependency statement. It notices that dependency lines for pgma.obj and pgmb.obj (the files on which pgm.exe depends) exist. So it processes those statements first.

2. Nmake.exe processes the pgma.obj dependency line. It notices that the pgma.obj file is newer than the pgma.asm file, so it does *not* execute the command following this dependency statement.

3. Nmake.exe processes the pgmb.obj dependency line. It notes that pgmb.obj is older than pgmb.asm (since we just changed the pgmb.asm source file). Therefore, nmake.exe executes the DOS command following on the next line. This generates a new pgmb.obj file that is now up to date.

4. Having process the pgma.obj and pgmb.obj dependencies, nmake.exe now returns its attention to the first dependency line. Since nmake.exe just created a new pgmb.obj file, its date/time stamp will be newer than pgm.exe's. Therefore, nmake.exe will execute the ml command that links pgma.obj and pgmb.obj together to form the new pgm.exe file.

Note that a properly written make file will instruct nmake.exe to assembly only those modules absolutely necessary to produce a consistent executable file. In the example above, nmake.exe did not bother to assemble pgma.asm since its object file was already up to date.

There is one final thing to emphasize with respect to dependencies. Often, object files are dependent not only on the source file that produces the object file, but any files that the source file includes as well. In the previous example, there (apparently) were no such include files. Often, this is not the case. A more typical make file might look like the following:

```
pgm.exe: pgma.obj pgmb.obj
       ml /Fepgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.asm pgm.a
       ml /c pgma.asm

pgmb.obj: pgmb.asm pgm.a
       ml /c pgmb.asm
```

Note that any changes to the pgm.a file will force nmake.exe to reassemble *both* pgma.asm and pgmb.asm since the pgma.obj and pgmb.obj files both depend upon the pgm.a include file. Leaving include files out of a dependency list is a common mistake programmers make that can produce inconsistent .exe files.

Note that you would not normally need to specify the UCR Standard Library include files nor the Standard Library .lib files in the dependency list. True, your resulting .exe file does depend on this code, but the Standard Library rarely changes, so you can safely leave it out of your dependency list. Should you make a modification to the Standard Library, simply delete any old .exe and .obj files and force a reassembly of the entire system.

Nmake.exe, by default, assumes that it will be processing a make file named "makefile". When you run nmake.exe, it looks for "makefile" in the current directory. If it doesn't find this file, it complains and terminates[18]. Therefore, it is a good idea to collect the files for each project you work on into their own subdirectory and give each project its own makefile. Then to create an executable, you need only change into the appropriate subdirectory and run the nmake.exe program.

Although this section discusses the nmake program in sufficient detail to handle most projects you will be working on, keep in mind that nmake.exe provides considerable functionality that this chapter does not discuss. To learn more about the nmake.exe program, consult the documentation that comes with MASM.

18. There is a command line option that lets you specify the name of the makefile. See the nmake documentation in the MASM manuals for more details.

8.22 Sample Program

Here is a single program that demonstrates most of the concepts from this chapter. This program consists of several files, including a makefile, that you can assemble and link using the nmake.exe program. This particular sample program computes "cross products" of various functions. The multiplication table you learned in school is a good example of a cross product, so are the truth tables found in Chapter Two of your textbook. This particular program generates cross product tables for addition, subtraction, division, and, optionally, remainder (modulo). In addition to demonstrating several concepts from this chapter, this sample program also demonstrates how to manipulate dynamically allocated arrays. This particular program asks the user to input the matrix size (row and column sizes) and then computes an appropriate set of cross products for that array.

8.22.1 EX8.MAK

The cross product program contains several modules. The following make file assembles all necessary files to ensure a consistent .EXE file.

```
ex8.exe:ex8.obj geti.obj getarray.obj xproduct.obj matrix.a
       ml ex8.obj geti.obj getarray.obj xproduct.obj

ex8.obj: ex8.asm matrix.a
       ml /c ex8.asm

geti.obj: geti.asm matrix.a
       ml /c geti.asm

getarray.obj: getarray.asm matrix.a
       ml /c getarray.asm


xproduct.obj: xproduct.asm matrix.a
       ml /c xproduct.asm
```

8.22.2 Matrix.A

MATRIX.A is the header file containing definitions that the cross product program uses. It also contains all the externdef statements for all externally defined routines.

```
; MATRIX.A
;
; This include file provides the external definitions
; and data type definitions for the matrix sample program
; in Chapter Eight.
;
; Some useful type definitions:

Integer          typedef  word
Char             typedef  byte

; Some common constants:

Bell             equ      07;ASCII code for the bell character.

; A "Dope Vector" is a structure containing information about arrays that
; a program allocates dynamically during program execution.  This particular
; dope vector handles two dimensional arrays.  It uses the following fields:
;
;      TTL-      Points at a zero terminated string containing a description
;                of the data in the array.
;
;      Func-     Pointer to function to compute for this matrix.
```

```
;
;       Data-     Pointer to the base address of the array.
;
;       Dim1-     This is a word containing the number of rows in the array.
;
;       Dim2-     This is a word containing the number of elements per row
;                 in the array.
;
;       ESize-    Contains the number of bytes per element in the array.

DopeVec           struct
TTL               dword     ?
Func              dword     ?
Data              dword     ?
Dim1              word      ?
Dim2              word      ?
ESize             word      ?
DopeVec           ends

; Some text equates the matrix code commonly uses:

Base              textequ   <es:[di]>

byp               textequ   <byte ptr>
wp                textequ   <word ptr>
dp                textequ   <dword ptr>

; Procedure declarations.

InpSeg            segment   para public 'input'

                  externdef geti:far
                  externdef getarray:far

InpSeg            ends


cseg              segment   para public 'code'

                  externdef CrossProduct:near

cseg              ends


; Variable declarations

dseg              segment   para public 'data'

                  externdef InputLine:byte

dseg              ends


; Uncomment the following equates if you want to turn on the
; debugging statements or if you want to include the MODULO function.

;debug            equ       0
;DoMOD            equ       0
```

8.22.3 EX8.ASM

This is the main program. It calls appropriate routines to get the user input, compute the cross product, and print the result.

```
; Sample program for Chapter Eight.
; Demonstrates the use of many MASM features discussed in Chapter Six
; including label types, constants, segment ordering, procedures, equates,
; address expressions, coercion and type operators, segment prefixes,
```

```
; the assume directive, conditional assembly, macros, listing directives,
; separate assembly, and using the UCR Standard Library.
;
; Include the header files for the UCR Standard Library.  Note that the
; “stdlib.a” file defines two segments; MASM will load these segments into
; memory before “dseg” in this program.
;
; The “.nolist” directive tells MASM not to list out all the macros for
; the standard library when producing an assembly listing.  Doing so would
; increase the size of the listing by many tens of pages and would tend to
; obscure the real code in this program.
;
; The “.list” directive turns the listing back on after MASM gets past the
; standard library files.  Note that these two directives (“.nolist” and
; “.list”) are only active if you produce an assembly listing using MASM’s
; “/Fl” command line parameter.


                .nolist
                include   stdlib.a
                includelib stdlib.lib
                .list



; The following statement includes the special header file for this
; particular program.  The header file contains external definitions
; and various data type definitions.

                include  matrix.a


; The following two statements allow us to use 80386 instructions
; in the program.  The “.386” directive turns on the 80386 instruction
; set, the “option” directive tells MASM to use 16-bit segments by
; default (when using 80386 instructions, 32-bit segments are the default).
; DOS real mode programs must be written using 16-bit segments.

                .386
                option   segment:use16



dseg            segment  para public ‘data’

Rows            integer  ?           ;Number of rows in matrices
Columns         integer  ?           ;Number of columns in matrices


; Input line is an input buffer this code uses to read a string of text
; from the user.  In particular, the GetWholeNumber procedure passes the
; address of InputLine to the GETS routine that reads a line of text
; from the user and places each character into this array.  GETS reads
; a maximum of 127 characters plus the enter key from the user.  It zero
; terminates that string (replacing the ASCII code for the ENTER key with
; a zero).  Therefore, this array needs to be at least 128 bytes long to
; prevent the possibility of buffer overflow.
;
; Note that the GetArray module also uses this array.

InputLine       char     128 dup (0)


; The following two pointers point at arrays of integers.
; This program dynamically allocates storage for the actual array data
; once the user tells the program how big the arrays should be.  The
; Rows and Columns variables above determine the respective sizes of
; these arrays.  After allocating the storage with a call to MALLOC,
; this program stores the pointers to these arrays into the following
; two pointer variables.
```

```
RowArray        dword   ?           ;Pointer to Row values
ColArray        dword   ?           ;Pointer to column values.


; ResultArrays is an array of dope vectors(*) to hold the results
; from the matrix operations:
;
; [0]- addition table
; [1]- subtraction table
; [2]- multiplication table
; [3]- division table
;
; [4]- modulo (remainder) table -- if the symbol "DoMOD" is defined.
;
; The equate that follows the ResultArrays declaration computes the number
; of elements in the array.  "$" is the offset into dseg immediately after
; the last byte of ResultArrays.  Subtracting this value from ResultArrays
; computes the number of bytes in ResultArrays.  Dividing this by the size
; of a single dope vector produces the number of elements in the array.
; This is an excellent example of how you can use address expressions in
; an assembly language program.
;
; The IFDEF DoMOD code demonstrates how easy it is to extend this matrix.
; Defining the symbol "DoMOD" adds another entry to this array.  The
; rest of the program adjusts for this new entry automatically.
;
; You can easily add new items to this array of dope vectors.  You will
; need to supply a title and a function to compute the matrice's entries.
; Other than that, however, this program automatically adjusts to any new
; entries you add to the dope vector array.
;
; (*) A "Dope Vector" is a data structure that describes a dynamically
; allocated array.  A typical dope vector contains the maximum value for
; each dimension, a pointer to the array data in memory, and some other
; possible information.  This program also stores a pointer to an array
; title and a pointer to an arithmetic function in the dope vector.

ResultArrays    DopeVec  {AddTbl,Addition}, {SubTbl,Subtraction}
                DopeVec  {MulTbl,Multiplication}, {DivTbl,Division}

                ifdef    DoMOD
                DopeVec  {ModTbl,Modulo}
                endif

; Add any new functions of your own at this point, before the following equate:


RASize          =        ($-ResultArrays) / (sizeof DopeVec)


; Titles for each of the four (five) matrices.

AddTbl          char     "Addition Table",0
SubTbl          char     "Subtraction Table",0
MulTbl          char     "Multiplication Table",0
DivTbl          char     "Division Table",0

                ifdef    DoMOD
ModTbl          char     "Modulo (Remainder) Table",0
                endif

; This would be a good place to put a title for any new array you create.

dseg            ends
```

```
; Putting PrintMat inside its own segment demonstrates that you can have
; multiple code segments within a program.  There is no reason we couldn't
; have put "PrintMat" in CSEG other than to demonstrate a far call to a
; different segment.

PrintSeg        segment  para public 'PrintSeg'

; PrintMat-     Prints a matrix for the cross product operation.
;
;               On Entry:
;
;                       DS must point at DSEG.
;                       DS:SI points at the entry in ResultArrays for the
;                       array to print.
;
; The output takes the following form:
;
;     Matrix Title
;
;             <- column matrix values ->
;
;     ^       *------------------------*
;     |       |                        |
;     R       |                        |
;     o       | Cross Product Matrix   |
;     w       |       Values           |
;             |                        |
;     V       |                        |
;     a       |                        |
;     l       |                        |
;     u       |                        |
;     e       |                        |
;     s       |                        |
;     |       |                        |
;     v       *------------------------*


PrintMat        proc    far
                assume  ds:dseg


; Note the use of conditional assembly to insert extra debugging statements
; if a special symbol "debug" is defined during assembly.  If such a symbol
; is not defined during assembly, the assembler ignores the following
; statements:

                ifdef   debug
                print
                char    "In PrintMat",cr,lf,0
                endif

; First, print the title of this table.  The TTL field in the dope vector
; contains a pointer to a zero terminated title string.  Load this pointer
; into es:di and call PUTS to print that string.

                putcr
                les     di, [si].DopeVec.TTL
                puts

; Now print the column values.  Note the use of PUTISIZE so that each
; value takes exactly six print positions. The following loop repeats
; once for each element in the Column array (the number of elements in
; the column array is given by the Dim2 field in the dope vector).

                print                           ;Skip spaces to move past the
                char    cr,lf,lf,"      ",0     ; row values.

                mov     dx, [si].DopeVec.Dim2   ;# times to repeat the loop.
                les     di, ColArray            ;Base address of array.
ColValLp:       mov     ax, es:[di]             ;Fetch current array element.
```

```
                mov     cx, 6                   ;Print the value using a
                putisize                        ; minimum of six positions.
                add     di, 2                   ;Move on to next element.
                dec     dx                      ;Repeat this loop DIM2 times.
                jne     ColValLp
                putcr                           ;End of column array output
                putcr                           ;Insert a blank line.

; Now output each row of the matrix.  Note that we need to output the
; RowArray value before each row of the matrix.
;
; RowLp is the outer loop that repeats for each row.

                mov     Rows, 0                 ;Repeat for 0..Dim1-1 rows.
RowLp:          les     di, RowArray            ;Output the current RowArray
                mov     bx, Rows                ; value on the left hand side
                add     bx, bx                  ; of the matrix.
                mov     ax, es:[di][bx]         ;ES:DI is base, BX is index.
                mov     cx, 5                   ;Output using five positions.
                putisize
                print
                char    ": ",0

; ColLp is the inner loop that repeats for each item on each row.

                mov     Columns, 0              ;Repeat for 0..Dim2-1 cols.
ColLp:          mov     bx, Rows                ;Compute index into the array
                imul    bx, [si].DopeVec.Dim2   ; index := (Rows*Dim2 +
                add     bx, Columns             ;              columns) * 2
                add     bx, bx

; Note that we only have a pointer to the base address of the array, so we
; have to fetch that pointer and index off it to access the desired array
; element.  This code loads the pointer to the base address of the array into
; the es:di register pair.

                les     di, [si].DopeVec.Data   ;Base address of array.
                mov     ax, es:[di][bx]         ;Get array element

; The functions that compute the values for the array store an 8000h into
; the array element if some sort of error occurs.  Of course, it is possible
; to produce 8000h as an actual result, but giving up a single value to
; trap errors is worthwhile.  The following code checks to see if an error
; occurred during the cross product.  If so, this code prints "  ****",
; otherwise, it prints the actual value.

                cmp     ax, 8000h               ;Check for error value
                jne     GoodOutput
                print
                char    "  ****",0              ;Print this for errors.
                jmp     DoNext

GoodOutput:     mov     cx, 6                   ;Use six print positions.
                putisize                        ;Print a good value.

DoNext:         mov     ax, Columns             ;Move on to next array
                inc     ax                      ; element.
                mov     Columns, ax
                cmp     ax, [si].DopeVec.Dim2   ;See if we're done with
                jb      ColLp                   ; this column.

                putcr                           ;End each column with CR/LF

                mov     ax, Rows                ;Move on to the next row.
                inc     ax
                mov     Rows, ax
                cmp     ax, [si].DopeVec.Dim1   ;Have we finished all the
                jb      RowLp                   ; rows?  Repeat if not done.
                ret
PrintMat        endp
```

```
PrintSeg        ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

;GetWholeNum-   This routine reads a whole number (an integer greater than
;               zero) from the user.  If the user enters an illegal whole
;               number, this procedure makes the user re-enter the data.

GetWholeNum     proc     near
                lesi     InputLine    ;Point es:di at InputLine array.
                gets

                call     Geti         ;Get an integer from the line.
                jc       BadInt       ;Carry set if error reading integer.
                cmp      ax, 0        ;Must have at least one row or column!
                jle      BadInt
                ret

BadInt:         print
                char     Bell
                char     "Illegal integer value, please re-enter",cr,lf,0
                jmp      GetWholeNum
GetWholeNum     endp


; Various routines to call for the cross products we compute.
; On entry, AX contains the first operand, dx contains the second.
; These routines return their result in AX.
; They return AX=8000h if an error occurs.
;
; Note that the CrossProduct function calls these routines indirectly.

addition        proc     far
                add      ax, dx
                jno      AddDone      ;Check for signed arithmetic overflow.
                mov      ax, 8000h    ;Return 8000h if overflow occurs.
AddDone:        ret
addition        endp


subtraction     proc     far
                sub      ax, dx
                jno      SubDone
                mov      ax, 8000h    ;Return 8000h if overflow occurs.
SubDone:        ret
subtraction     endp

multiplication   proc    far
                 imul    ax, dx
                 jno     MulDone
                 mov     ax, 8000h    ;Error if overflow occurs.
MulDone:         ret
multiplication   endp

division        proc     far
                push     cx           ;Preserve registers we destory.

                mov      cx, dx
                cwd
                test     cx, cx       ;See if attempting division by zero.
                je       BadDivide
                idiv     cx

                mov      dx, cx       ;Restore the munged register.
                pop      cx
                ret

BadDivide:      mov      ax, 8000h
```

```
                mov     dx, cx
                pop     cx
                ret
division        endp


; The following function computes the remainder if the symbol "DoMOD"
; is defined somewhere prior to this point.

                ifdef   DoMOD
modulo          proc    far
                push    cx

                mov     cx, dx
                cwd
                test    cx, cx      ;See if attempting division by zero.
                je      BadDivide
                idiv    cx
                mov     ax, dx      ;Need to put remainder in AX.
                mov     dx, cx      ;Restore the munged registers.
                pop     cx
                ret

BadMod:         mov     ax, 8000h
                mov     dx, cx
                pop     cx
                ret
modulo          endp
                endif

; If you decide to extend the ResultArrays dope vector array, this is a good
; place to define the function for those new arrays.


; The main program that reads the data from the user, calls the appropriate
; routines, and then prints the results.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; Prompt the user to enter the number of rows and columns:

GetRows:        print
                byte    "Enter the number of rows for the matrix:",0

                call    GetWholeNum
                mov     Rows, ax

; Okay, read each of the row values from the user:

                print
                char    "Enter values for the row (vertical) array",cr,lf,0

; Malloc allocates the number of bytes specified in the CX register.
; AX contains the number of array elements we want;  multiply this value
; by two since we want an array of words.  On return from malloc, es:di
; points at the array allocated on the "heap".  Save away this pointer in
; the "RowArray" variable.
;
; Note the use of the "wp" symbol. This is an equate to "word ptr" appearing
; in the "matrix.a" include file.  Also note the use of the address expression
; "RowArray+2" to access the segment portion of the double word pointer.

                mov     cx, ax
                shl     cx, 1
                malloc
                mov     wp RowArray, di
```

```
                mov     wp RowArray+2, es

; Okay, call "GetArray" to read "ax" input values from the user.
; GetArray expects the number of values to read in AX and a pointer
; to the base address of the array in es:di.

                print
                char    "Enter row data:",0

                mov     ax, Rows    ;# of values to read.
                call    GetArray    ;ES:DI still points at array.


; Okay, time to repeat this for the column (horizontal) array.

GetCols:        print
                byte    "Enter the number of columns for the matrix:",0

                call    GetWholeNum ;Get # of columns from the user.
                mov     Columns, ax ;Save away number of columns.


; Okay, read each of the column values from the user:

                print
                char    "Enter values for the column (horz.) array",cr,lf,0

; Malloc allocates the number of bytes specified in the CX register.
; AX contains the number of array elements we want;  multiply this value
; by two since we want an array of words.  On return from malloc, es:di
; points at the array allocated on the "heap".  Save away this pointer in
; the "RowArray" variable.

                mov     cx, ax                  ;Convert # Columns to # bytes
                shl     cx, 1                   ; by multiply by two.
                malloc                          ;Get the memory.
                mov     wp ColArray, di         ;Save pointer to the
                mov     wp ColArray+2, es       ;columns vector (array).

; Okay, call "GetArray" to read "ax" input values from the user.
; GetArray expects the number of values to read in AX and a pointer
; to the base address of the array in es:di.

                print
                char    "Enter Column data:",0

                mov     ax, Columns ;# of values to read.
                call    GetArray    ;ES:DI points at column array.

; Okay, initialize the matrices that will hold the cross products.
; Generate RASize copies of the following code.
; The "repeat" macro repeats the statements between the "repeat" and the "endm"
; directives RASize times.  Note the use of the Item symbol to automatically
; generate different indexes for each repetition of the following code.
; The "Item = Item+1" statement ensures that Item will take on the values
; 0, 1, 2, ..., RASize on each repetition of this loop.
;
; Remember, the "repeat..endm" macro copies the statements multiple times
; within the source file, it does not execute a "repeat..until" loop at
; run time.  That is, the following macro is equivalent to making "RASize"
; copies of the code, substituting different values for Item for each
; copy.
;
; The nice thing about this code is that it automatically generates the
; proper amount of initialization code, regardless of the number of items
; placed in the ResultArrays array.


Item            =       0
```

```
                repeat  RASize

                mov     cx, Columns  ;Compute the size, in bytes,
                imul    cx, Rows     ; of the matrix and allocate
                add     cx, cx       ; sufficient storage for the
                malloc               ; array.

                mov     wp ResultArrays[Item * (sizeof DopeVec)].Data, di
                mov     wp ResultArrays[Item * (sizeof DopeVec)].Data+2, es

                mov     ax, Rows
                mov     ResultArrays[Item * (sizeof DopeVec)].Dim1, ax

                mov     ax, Columns
                mov     ResultArrays[Item * (sizeof DopeVec)].Dim2, ax

                mov     ResultArrays[Item * (sizeof DopeVec)].ESize, 2

Item            =       Item+1
                endm


; Okay, we've got the input values from the user,
; now let's compute the addition, subtraction, multiplication,
; and division tables.  Once again, a macro reduces the amount of
; typing we need to do at this point as well as automatically handling
; however many items are present in the ResultArrays array.

element         =       0

                repeat  RASize
                lfs     bp, RowArray ;Pointer to row data.
                lgs     bx, ColArray ;Pointer to column data.

                lea     cx, ResultArrays[element * (sizeof DopeVec)]
                call    CrossProduct

element         =       element+1
                endm


; Okay, print the arrays down here.  Once again, note the use of the
; repeat..endm macro to save typing and automatically handle additions
; to the ResultArrays array.


Item            =       0

                repeat  RASize
                mov     si, offset ResultArrays[item * (sizeof DopeVec)]
                call    PrintMat
Item            =       Item+1
                endm


; Technically, we don't have to free up the storage malloc'd for each
; of the arrays since the program is about to quit.  However, it's a
; good idea to get used to freeing up all your storage when you're done
; with it.  For example, were you to add code later at the end of this
; program, you would have that extra memory available to that new code.

                les     di, ColArray
                free
                les     di, RowArray
                free

Item            =       0
                repeat  RASize
                les     di, ResultArrays[Item * (sizeof DopeVec)].Data
                free
```

```
Item            =       Item+1
                endm


Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

8.22.4 GETI.ASM

GETI.ASM contains a routine (geti) that reads an integer value from the user.

```
; GETI.ASM
;
; This module contains the integer input routine for the matrix
; example in Chapter Eight.

                .nolist
                include  stdlib.a
                .list

                include  matrix.a

InpSeg          segment  para public 'input'

; Geti-On entry, es:di points at a string of characters.
;      This routine skips any leading spaces and comma characters and then
;      tests the first (non-space/comma) character to see if it is a digit.
;      If not, this routine returns the carry flag set denoting an error.
;      If the first character is a digit, then this routine calls the
;      standard library routine "atoi2" to convert the value to an integer.
;      It then ensures that the number ends with a space, comma, or zero
;      byte.
;
;      Returns carry clear and value in AX if no error.
;      Returns carry set if an error occurs.
;
;      This routine leaves ES:DI pointing at the character it fails on when
;      converting the string to an integer.  If the conversion occurs without
;      an error, the ES:DI points at a space, comma, or zero terminating byte.


geti            proc     far

                ifdef    debug
                print
                char     "Inside GETI",cr,lf,0
                endif

; First, skip over any leading spaces or commas.
; Note the use of the "byp" symbol to save having to type "byte ptr".
; BYP is a text equate appearing in the macros.a file.
; A "byte ptr" coercion operator is required here because MASM cannot
; determine the size of the memory operand (byte, word, dword, etc)
; from the operands.  I.e., "es:[di]" and ' ' could be any of these
; three sizes.
;
; Also note a cute little trick here; by decrementing di before entering
```

```
; the loop and then immediately incrementing di, we can increment di before
; testing the character in the body of the loop.  This makes the loop
; slightly more efficient and a lot more elegant.

                dec     di
SkipSpcs:       inc     di
                cmp     byp es:[di], ' '
                je      SkipSpcs
                cmp     byp es:[di], ','
                je      SkipSpcs

; See if the first non-space/comma character is a decimal digit:

                mov     al, es:[di]
                cmp     al, '-'       ;Minus sign is also legal in integers.
                jne     TryDigit
                mov     al, es:[di+1] ;Get next char, if "-"

TryDigit:       isdigit
                jne     BadGeti     ;Jump if not a digit.

; Okay, convert the characters that follow to an integer:

ConvertNum:     atoi2               ;Leaves integer in AX
                jc      BadGeti     ;Bomb if illegal conversion.

; Make sure this number ends with a reasonable character (space, comma,
; or a zero byte):

                cmp     byp es:[di], ' '
                je      GoodGeti
                cmp     byp es:[di], ','
                je      GoodGeti
                cmp     byp es:[di], 0
                je      GoodGeti

                ifdef   debug
                print
                char    "GETI: Failed because number did not end with "
                char    "a space, comma, or zero byte",cr,lf,0
                endif

BadGeti:        stc                 ;Return an error condition.
                ret

GoodGeti:       clc                 ;Return no error and an integer in AX
                ret
geti            endp

InpSeg          ends
                end
```

8.22.5 GetArray.ASM

GetArray.ASM contains the GetArray input routine. This reads the data for the array from the user to produce the cross products. Note that GetArray reads the data for a single dimension array (or one row in a multidimensional array). The cross product program reads two such vectors: one for the column values and one for the row values in the cross product. Note: This routine uses subroutines from the UCR Standard Library that appear in the next chapter.

```
; GETARRAY.ASM
;
; This module contains the GetArray input routine.  This routine reads a
; set of values for a row of some array.

                .386
```

```
                option  segment:use16

                .nolist
                include stdlib.a
                .list

                include matrix.a

; Some local variables for this module:

localdseg       segment para public 'LclData'

NumElements     word    ?
ArrayPtr        dword   ?

Localdseg       ends


InpSeg          segment para public 'input'
                assume  ds:Localdseg

; GetArray-     Read a set of numbers and store them into an array.
;
;               On Entry:
;
;                       es:di points at the base address of the array.
;                       ax contains the number of elements in the array.
;
;               This routine reads the specified number of array elements
;               from the user and stores them into the array.  If there
;               is an input error of some sort, then this routine makes
;               the user reenter the data.

GetArray        proc    far
                pusha               ;Preserve all the registers
                push    ds          ; that this code modifies
                push    es
                push    fs

                ifdef   debug
                print
                char    "Inside GetArray, # of input values =",0
                puti
                putcr
                endif

                mov     cx, Localdseg           ;Point ds at our local
                mov     ds, cx                  ; data segment.

                mov     wp ArrayPtr, di         ;Save in case we have an
                mov     wp ArrayPtr+2, es       ; error during input.
                mov     NumElements, ax

; The following loop reads a line of text from the user containing some
; number of integer values.  This loop repeats if the user enters an illegal
; value on the input line.
;
; Note: LESI is a macro from the stdlib.a include file.  It loads ES:DI
; with the address of its operand (as opposed to les di, InputLine that would
; load ES:DI with the dword value at address InputLine).

RetryLp:        lesi    InputLine               ;Read input line from user.
                gets
                mov     cx, NumElements         ;# of values to read.
                lfs     si, ArrayPtr            ;Store input values here.

; This inner loop reads "ax" integers from the input line.  If there is
; an error, it transfers control to RetryLp above.

ReadEachItem:   call    geti        ;Read next available value.
```

```
                jc      BadGA
                mov     fs:[si], ax  ;Save away in array.
                add     si, 2        ;Move on to next element.
                loop    ReadEachItem ;Repeat for each element.

                pop     fs           ;Restore the saved registers
                pop     es           ; from the stack before
                pop     ds           ; returning.
                popa
                ret

; If an error occurs, make the user re-enter the data for the entire
; row:

BadGA:          print
                char    "Illegal integer value(s).",cr,lf
                char    "Re-enter data:",0
                jmp     RetryLp
getArray        endp

InpSeg          ends
                end
```

8.22.6 XProduct.ASM

This file contains the code that computes the actual cross-product.

```
; XProduct.ASM-
;
;       This file contains the cross-product module.

                .386
                option    segment:use16

                .nolist
                include   stdlib.a
                includelib stdlib.lib
                .list

                include   matrix.a

; Local variables for this module.

dseg            segment  para public 'data'
DV              dword    ?
RowNdx          integer  ?
ColNdx          integer  ?
RowCntr         integer  ?
ColCntr         integer  ?
dseg            ends


cseg            segment  para public 'code'
                assume   ds:dseg

; CrossProduct- Computes the cartesian product of two vectors.
;
;                       On entry:
;
;                       FS:BP-Points at the row matrix.
;                       GS:BX-Points at the column matrix.
;                       DS:CX-Points at the dope vector for the destination.
;
;               This code assume ds points at dseg.
;               This routine only preserves the segment registers.

RowMat          textequ  <fs:[bp]>
ColMat          textequ  <gs:[bx]>
```

```
DVP             textequ  <ds:[bx].DopeVec>

CrossProduct    proc     near

                ifdef    debug
                print
                char     "Entering CrossProduct routine",cr,lf,0
                endif

                xchg     bx, cx       ;Get dope vector pointer
                mov      ax, DVP.Dim1 ;Put Dim1 and Dim2 values
                mov      RowCntr, ax  ; where they are easy to access.
                mov      ax, DVP.Dim2
                mov      ColCntr, ax
                xchg     bx, cx


; Okay, do the cross product operation.  This is defined as follows:
;
;      for RowNdx := 0 to NumRows-1 do
;          for ColNdx := 0 to NumCols-1 do
;                Result[RowNdx, ColNdx] = Row[RowNdx] op Col[ColNdx];

                mov      RowNdx, -1   ;Really starts at zero.
OutsideLp:      add      RowNdx, 1
                mov      ax, RowNdx
                cmp      ax, RowCntr
                jge      Done

                mov      ColNdx, -1   ;Really starts at zero.
InsideLp:       add      ColNdx, 1
                mov      ax, ColNdx
                cmp      ax, ColCntr
                jge      OutSideLp

                mov      di, RowNdx
                add      di, di
                mov      ax, RowMat[di]

                mov      di, ColNdx
                add      di, di
                mov      dx, ColMat[di]

                push     bx           ;Save pointer to column matrix.
                mov      bx, cx       ;Put ptr to dope vector where we can
                                      ; use it.

                call    DVP.Func      ;Compute result for this guy.

                mov      di, RowNdx   ;Index into array is
                imul     di, DVP.Dim2 ; (RowNdx*Dim2 + ColNdx) * ElementSize
                add      di, ColNdx
                imul     di, DVP.ESize

                les      bx, DVP.Data           ;Get base address of array.
                mov      es:[bx][di], ax        ;Save away result.

                pop      bx                     ;Restore ptr to column array.
                jmp      InsideLp

Done:           ret
CrossProduct    endp
cseg            ends
                end
```

8.23 Laboratory Exercises

In this set of laboratory exercises you will assemble various short programs, produce assembly listings, and observe the object code the assembler produces for some simple instruction sequences. You will also experiment with a make file to observe how it properly handles dependencies.

8.23.1 Near vs. Far Procedures

The following short program demonstrates how MASM automatically generates near and far call and ret instructions depending on the operand field of the proc directive (this program is on the companion CD-ROM in the chapter eight subdirectory).

Assemble this program with the /Fl option to produce an assembly listing. Look up the opcodes for near and far call and ret instructions in Appendix D. Compare those values against the opcodes this program emits. **For your lab report**: describe how MASM figures out which instructions need to be near or far. Include the assembled listing with your report and identify which instructions are near or far calls and returns.

```
; EX8_1.asm (Laboratory Exercise 8.1)

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Procedure1      proc     near

; MASM will emit a *far* call to procedure2
; since it is a far procedure.

                call     Procedure2

; Since this return instruction is inside
; a near procedure, MASM will emit a near
; return.

                ret
Procedure1      endp


Procedure2      proc     far

; MASM will emit a *near* call to procedure1
; since it is a near procedure.

                call     Procedure1

; Since this return instruction is inside
; a far procedure, MASM will emit a far
; return.

                ret
Procedure2      endp

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; MASM emits the appropriate call instructions
; to the following procedures.

                call     Procedure1
                call     Procedure2

Quit:           mov      ah, 4ch
```

```
                int     21h
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends
                end      Main
```

8.23.2 Data Alignment Exercises

In this exercise you will compile two different programs using the MASM "/Fl" command line option so you can observe the addresses MASM assigns to the variables in the program. The first program (Ex8_2a.asm) uses the even directive to align objects on a word boundary. The second program (Ex8_2b.asm) uses the align directive to align objects on different sized boundaries. **For your lab report:** Include the assembly listings in your lab report. Describe what the even and align directives are doing in the program and comment on how this produces faster running programs.

```
; EX8_2a.asm
;
; Example demonstrating the EVEN directive.

dseg            segment

; Force an odd location counter within
; this segment:

i               byte     0

; This word is at an odd address, which is bad!

j               word     0

; Force the next word to align itself on an
; even address so we get faster access to it.

                even
k               word     0

; Note that even has no effect if we're already
; at an even address.

                even
l               word     0
dseg            ends

cseg            segment
                assume   ds:dseg
procedure       proc
                mov      ax, [bx]
                mov     i, al
                mov      bx, ax

; The following instruction would normally lie on
; an odd address.  The EVEN directive inserts a
; NOP so that it falls on an even address.

                even
                mov      bx, cx

; Since we're already at an even address, the
; following EVEN directive has no effect.

                even
                mov      dx, ax
```

```
                ret
procedure       endp
cseg            ends
                end




; EX8_2b.asm
;
; Example demonstrating the align
; directive.

dseg            segment

; Force an odd location counter
; within this segment:

i               byte    0

; This word is at an odd address,
; which is bad!

j               word    0

; Force the next word to align itself
; on an even address so we get faster
; access to it.

                align   2
k               word    0

; Force odd address again:

k_odd           byte    0

; Align the next entry on a double
; word boundary.

                align   4
l               dword   0

; Align the next entry on a quad
; word boundary:

                align   8
RealVar         real8   3.14159

; Start the following on a paragraph
; boundary:

                align   16
Table           dword   1,2,3,4,5
dseg            ends
                end
```

8.23.3 Equate Exercise

In this exercise you will discover a major difference between a numeric equate and a textual equate (program Ex8_3.asm on the companion CD-ROM). MASM evaluates the operand field of a numeric equate when it encounters the equate. MASM defers evaluation of a textual equate until it expands the equate (i.e., when you use the equate in a program). **For your lab report:** assemble the following program using MASM's "/Fl" command line option and look at the object code emitted for the two equates. Explain

why the instruction operands are different even though the two equates are nearly identical.

```
; Ex8_3.asm
;
; Comparison of numeric equates with textual equates
; and the differences they produce at assembly time.
;
cseg            segment
equ1            equ     $+2         ;Evaluates "$" at this stmt.
equ2            equ     <$+2>       ;Evaluates "$" on use.
MyProc          proc
                mov     ax, 0
                lea     bx, equ1
                lea     bx, equ2
                lea     bx, equ1
                lea     bx, equ2
MyProc          endp
cseg            ends
                end
```

8.23.4 IFDEF Exercise

In this exercise, you will assemble a program that uses conditional assembly and observe the results. The Ex8_4.asm program uses the ifdef directive to test for the presence of DEBUG1 and DEBUG2 symbols. DEBUG1 appears in this program while DEBUG2 does not. **For your lab report:** assemble this code using the "/Fl" command line parameter. Include the listing in your lab report and explain the actions of the ifdef directives.

```
; Ex8_4.asm
;
; Demonstration of IFDEF to control
; debugging features.  This code
; assumes there are two levels of
; debugging controlled by the two
; symbols DEBUG1 and DEBUG2.  In
; this code example DEBUG1 is
; defined while DEBUG2 is not.

                .xlist
                include  stdlib.a
                .list
                .nolistmacro
                .listif

DEBUG1          =       0

cseg            segment
DummyProc       proc
                ifdef    DEBUG2
                print
                byte     "In DummyProc"
                byte     cr,lf,0
                endif
                ret
DummyProc       endp

Main            proc
                ifdef    DEBUG1
                print
                byte     "Calling DummyProc"
                byte     cr,lf,0
                endif

                call     DummyProc

                ifdef    DEBUG1
```

```
                print
                byte    "Return from DummyProc"
                byte    cr,lf,0
                endif
                ret
Main            endp
cseg            ends
                end
```

8.23.5 Make File Exercise

In this exercise you will experiment with a make file to see how nmake.exe chooses which files to reassemble. In this exercise you will be using the Ex8_5a.asm, Ex8_5b.asm, Ex8_5.a, and Ex8_5.mak files found in the Chapter Eight subdirectory on the companion CD-ROM. Copy these files to a local subdirectory on your hard disk (if they are not already there). These files contain a program that reads a string of text from the user and prints out any vowels in the input string. You will make minor changes to the .asm and .a files and run the make file and observe the results.

The first thing you should do is assemble the program and create up to date .exe and .obj files for the project. You can do this with the following DOS command:

```
nmake Ex8_5.mak
```

Assuming that the .obj and .exe files were not already present in the current directory, the nmake command above will assemble and link the Ex8_5a.asm and Ex8_5b.asm files producing the Ex8.exe executable.

Using the editor, make a minor change (such as inserting a single space on a line containing a comment) to the Ex8_5a.asm file. Execute the above nmake command. Record what the make file does in your lab report.

Next, make a minor change to the Ex8_5b.asm file. Run the above nmake command and record the result in your lab report. Explain the results.

Finally, make a minor change to the Ex8_5.a file. Run the nmake command and describe the results in your lab report.

For your lab report: explain how the changes to each of the files above affects the make operation. Explain why nmake does what it does. **For additional credit:** Try deleting (one at a time) the Ex8_5a.obj, Ex8_5b.obj, and Ex8_5.exe files and run the nmake command. Explain why nmake does what it does when you individually delete each of these files.

Ex8_5.mak makefile:

```
ex8_5.exe: ex8_5a.obj ex8_5b.obj
    ml /Feex8_5.exe ex8_5a.obj ex8_5b.obj

ex8_5a.obj: ex8_5a.asm ex8_5.a
    ml /c ex8_5a.asm

ex8_5b.obj: ex8_5b.asm ex8_5.a
    ml /c ex8_5b.asm
```

Ex8_5.a Header File:

```
; Header file for Ex8_5 project.
; This file includes the EXTERNDEF
; directive which makes the PrintVowels
; name public/external.  It also includes
; the PrtVowels macro which lets us call
; the PrintVowels routine in a manner
; similar to the UCR Standard Library
```

```
; routines.

                externdef PrintVowels:near

PrtVowels       macro
                call    PrintVowels
                endm
```

Ex8_5a.asm source file:

```
; Ex8_5a.asm
;
; Randall Hyde
; 2/7/96
;
; This program reads a string of symbols from the
; user and prints the vowels. It demonstrates the use of
; make files


                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

; The following include file brings in the external
; definitions of the routine(s) in the Lab6x10b
; module.  In particular, it gives this module
; access to the "PrtVowels" routine found in
; Lab8_5b.asm.

                include    Ex8_5.a


cseg            segment  para public 'code'

Main            proc

                meminit

; Read a string from the user, print all the vowels
; present in that string, and then free up the memory
; allocated by the GETSM routine:

                print
                byte     "I will find all your vowels"
                byte     cr,lf
                byte     "Enter a line of text: ",0

                getsm
                print
                byte     "Vowels on input line: ",0
                PrtVowels
                putcr
                free

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
```

```
zzzzzzseg       ends
                end     Main
```

8.24 Programming Projects

1) Write a program that inputs two 4x4 integer matrices from the user and compute their matrix product. The matrix multiply algorithm (computing C := A * B) is

```
for i := 0 to 3 do
          for j := 0 to 3 do begin

                    c[i,j] := 0;
                    for k := 0 to 3 do
                        c[i,j] := c[i,j] + a[i,k] * b[k,j];
          end;
```

Feel free to use the ForLp and Next macros from Chapter Six.

2) Modify the sample program (“Sample Program” on page 432) to use the FORLP and NEXT macros provided in the textbook. Replace all for loop simulations in the program with the corresponding macros.

3) Write a program that asks the user to input three integer values, m, p, and n. This program should allocate storage for three arrays: A[0..m-1, 0..p-1], B[0..p-1, 0..n-1], and C[0..m-1, 0..n-1]. The program should then read values for arrays A and B from the user. Next, this program should compute the matrix product of A and B using the algorithm:

```
for i := 0 to m-1 do
          for j := 0 to n-1 do begin

                    c[i,j] := 0;
                    for k := 0 to p-1 do
                        c[i,j] := c[i,j] + a[i,k] * b[k,j];
          end;
```

Finally, the program should print arrays A, B, and C. Feel free to use the ForLp and Next macro given in this chapter. You should also take a look at the sample program (see “Sample Program” on page 432) to see how to dynamically allocate storage for arrays and access arrays whose dimensions are not known until run time.

4) The ForLp and Next macros provide in this chapter only increment their loop control variable by one on each iteration of the loop. Write a new macro, ForTo, that lets you specify an *increment* constant. Increment the loop control variable by this constant on each iteration of the for loop. Write a program to demonstrate the use of this macro. Hint: you will need to create a global label to pass the increment information to the NEXT macro, or you will need to perform the increment operation inside the ForLp macro.

5) Write a third version for ForLp and Next (see Program #7 above) that lets you specify *negative* increments (like the for..downto statement in Pascal). Call this macro ForDT (for..downto).

8.25 Summary

This chapter introduced several assembler directives and pseudo-opcodes supported by MASM. This chapter, by no means, is a complete description of what MASM has to offer. It does provide enough information to get you going.

Assembly language statements are free format and there is usually one statement per line in your source file. Although MASM allows free format input, you should carefully structure your source files to make them easier to read.

- See “Assembly Language Statements” on page 355.

MASM keeps track of the offset of an instruction or variable in a segment using the *location counter*. MASM increments the location counter by one for each byte of object code it writes to the output file.

- See “The Location Counter” on page 357.

Like HLLs, MASM lets you use symbolic names for variables and statement labels. Dealing with symbols is much easier than numeric offsets in an assembly language program. MASM symbols look a whole lot like their HLL with a few extensions.

- See “Symbols” on page 358

MASM provides several different types of literal constants including binary, decimal, and hexadecimal integer constants, string constants, and text constants.

- See “Literal Constants” on page 359.
- See “Integer Constants” on page 360.
- See “String Constants” on page 361.
- See “Text Constants” on page 362.

To help you manipulate segments within your program, MASM provides the segment/ends directives. With the segment directive you can control the loading order and alignment of modules in memory.

- See “Segments” on page 366.
- See “Segment Names” on page 367.
- See “Segment Loading Order” on page 368.
- See “Segment Operands” on page 369.
- See “The CLASS Type” on page 374.
- See “Typical Segment Definitions” on page 376.
- See “Why You Would Want to Control the Loading Order” on page 376.

MASM provides the proc/endp directives for declaring procedures within your assembly language programs. Although not strictly necessary, the proc/endp directives make your programs much easier to read and maintain. The proc/endp directives also let you use local statement names within your procedures.

- See “Procedures” on page 365.

Equates let you define symbolic constants of various sorts in your program. MASM provides three directives for defining such constants: “=”, equ, and textequ. As with HLLs, the judicious use of equates can help make your program easier to read.

- See “Declaring Manifest Constants Using Equates” on page 362.

As you saw in Chapter Four, MASM gives you the ability to declare variables in the data segment using the byte, word, dword and other directives. MASM is a strongly typed assembler and attaches a type as well as a location to variable names (most assemblers only attach a location). This helps MASM locate obscure bugs in your program.

- See “Variables” on page 384.
- See“Label Types” on page 385.
- See “How to Give a Symbol a Particular Type” on page 385.
- See “Label Values” on page 386.
- See “Type Conflicts” on page 386.

MASM supports *address expressions* that let you use arithmetic operators to build constant address values at assembly time. It also lets you override the type of an address value and extract various pieces of information about a symbol. This is very useful for writing maintainable programs.

- See “Address Expressions” on page 387.
- See “Symbol Types and Addressing Modes” on page 387.
- See “Arithmetic and Logical Operators” on page 388.
- See “Coercion” on page 390.
- See “Type Operators” on page 392.

- See “Operator Precedence” on page 396.

MASM provides several facilities for telling the assembler which segment associates with which segment register. It also gives you the ability to override a default choice. This lets your program manage several segments at once with a minimum of fuss.

- See “Segment Prefixes” on page 377.
- See “Controlling Segments with the ASSUME Directive” on page 377.

MASM provides you with a “conditional assembly” capability that lets you choose which segments of code are actually assembled during the assembly process. This is useful for inserting debugging code into your programs (that you can easily remove with a single statement) and for writing programs that need to run in different environments (by inserting and removing different sections of code).

- See “Conditional Assembly” on page 397.
- See “IF Directive” on page 398.
- See “IFE directive” on page 399.
- See “IFDEF and IFNDEF” on page 399.
- See “IFB, IFNB” on page 399.
- See “IFIDN, IFDIF, IFIDNI, and IFDIFI” on page 400.

MASM, living up to its name, provides a powerful macro facility. Macros are sections of code you can replicate by simply placing the macro’s name in your code. Macros, properly used, can help you write shorter, easier to read, and more robust programs. Alas, improperly used, macros produce hard to maintain, inefficient programs.

- See “Macros” on page 400.
- See “Procedural Macros” on page 400.
- See “The LOCAL Directive” on page 406.
- See“The EXITM Directive” on page 406.
- See “Macros: Good and Bad News” on page 419.
- See “Repeat Operations” on page 420.

MASM provides several directives you can use to produce “assembled listings” or print-outs of your program with lots of assembler generated (useful!) information. These directives let you turn on and off the listing operation, display information on the display during assembly, and set titles on the output.

- See “Controlling the Listing” on page 424.
- See “The ECHO and %OUT Directives” on page 424.
- See “The TITLE Directive” on page 424.
- See “The SUBTTL Directive” on page 424.
- See “The PAGE Directive” on page 424.
- See “The .LIST, .NOLIST, and .XLIST Directives” on page 425.
- See “Other Listing Directives” on page 425.

To handle large projects (“Programming in the Large”) requires separate compilation (or separate assembly in MASM’s case). MASM provides several directives that let you merge source files during assembly, separately assemble modules, and communicate procedure and variables names between the modules.

- See “Managing Large Programs” on page 425.
- See “The INCLUDE Directive” on page 426.
- See “The PUBLIC, EXTERN, and EXTRN Directives” on page 427.
- See “The EXTERNDEF Directive” on page 428.

8.26 Questions

1) What is the difference between the following instruction sequences?

```
        MOV     AX, VAR+1

and     MOV     AX, VAR
        INC     AX
```

2) What is the source line format for an assembly language statement?

3) What is the purpose of the ASSUME directive?

4) What is the location counter?

5) Which of the following symbols are valid?

a) ThisIsASymbol
b) This_Is_A_Symbol
c) This.Is.A.Symbol
d) .Is_This_A_Symbol?
e) _____
f) @_$?_To_You
g) 1WayToGo
h) %Hello
i) F000h
j) ?A_0$1
k) $1234
l) Hello there

6) How do you specify segment loading order?

7) What is the type of the symbols declared by the following statements?

```
a)symbol1       equ     0
b)symbol2:
c)symbol3       proc
d)symbol4       db      ?
e)symbol5       dw      ?
f)symbol6       proc    far
g)symbol7       equ     this word
h)symbol8       equ     byte ptr symbol7
i)symbol9       dd      ?
j)symbol10      macro
k)symbol11      segment para public 'data'
l)symbol12      equ     this near
m)symbol13      equ     'ABCD'
n)symbol14      equ     <MOV AX, 0>
```

8) Which of the symbols in question 7 are not assigned the current location counter value?

9) Explain the purpose of the following operators:

a) PTR b) SHORT c) THIS d) HIGH e) LOW

f) SEG g) OFFSET

10) What is the difference between the values loaded into the BX register (if any) in the following code sequence?

```
        mov     bx, offset Table
        lea     bx, Table
```

11) What is the difference between the REPEAT macro and the DUP operator?

12) In what order will the following segments be loaded into memory?

```
CSEG            segment  para public 'CODE'
                …
CSEG            ends
DSEG            segment  para public 'DATA'
                …
DSEG            ends
ESEG            segment  para public 'CODE'
                …
```

```
ESEG            ends
```

13) Which of the following address expressions do not produce the same result as the others:

a) Var1[3][5] b) 15[Var1] c) Var1[8] d) Var1+2[6]

e) Var1*3*5 f) Var1+3+5

Arithmetic and Logical Operations Chapter Nine

There is a lot more to assembly language than knowing the operations of a handful of machine instructions. You've got to know how to use them and what they can do. Many instructions are useful for operations that have little to do with their mathematical or obvious functions. This chapter discusses how to convert expressions from a high level language into assembly language. It also discusses advanced arithmetic and logical operations including multiprecision operations and tricks you can play with various instructions.

9.0 Chapter Overview

This chapter discusses six main subjects: converting HLL arithmetic expressions into assembly language, logical expressions, extended precision arithmetic and logical operations, operating on different sized operands, machine and arithmetic idioms, and masking operations. Like the preceding chapters, this chapter contains considerable material that you may need to learn immediately if you're a beginning assembly language programmer. The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- Arithmetic expressions
- Simple assignments
- Simple expressions
- Complex expressions
- Commutative operators
- Logical expressions
- Multiprecision operations
- Multiprecision addition operations
- Multiprecision subtraction operations
- Extended precision comparisons

❑ Extended precision multiplication
❑ Extended precision division
❑ Extended precision negation

- Extended precision AND, OR, XOR, and NOT

❑ Extended precision shift and rotate operations
❑ Operating on different sized operands

- Multiplying without MUL and IMUL

❑ Division without DIV and IDIV
❑ Using AND to compute remainders
❑ Modulo-n Counters with AND
❑ Testing for 0FFFFF...FFFh

- Test operations

❑ Testing signs with the XOR instructions
❑ Masking operations
❑ Masking with the AND instructions
❑ Masking with the OR instruction
❑ Packing and unpacking data types
❑ Table lookups

None of this material is particularly difficult to understand. However, there are a lot of new topics here and taking them a few at a time will certain help you absorb the material better. Those topics with the "•" prefix are ones you will frequently use; hence it is a good idea to study these first.

9.1 Arithmetic Expressions

Probably the biggest shock to beginners facing assembly language for the very first time is the lack of familiar arithmetic expressions. Arithmetic expressions, in most high level languages, look similar to their algebraic equivalents:

```
X:=Y*Z;
```

In assembly language, you'll need several statements to accomplish this same task, e.g.,

```
mov     ax, y
imul    z
mov     x, ax
```

Obviously the HLL version is much easier to type, read, and understand. This point, more than any other, is responsible for scaring people away from assembly language.

Although there is a lot of typing involved, converting an arithmetic expression into assembly language isn't difficult at all. By attacking the problem in steps, the same way you would solve the problem by hand, you can easily break down any arithmetic expression into an equivalent sequence of assembly language statements. By learning how to convert such expressions to assembly language in three steps, you'll discover there is little difficulty to this task.

9.1.1 Simple Assignments

The easiest expressions to convert to assembly language are the simple assignments. Simple assignments copy a single value into a variable and take one of two forms:

```
variable := constant
```

or

```
variable := variable
```

If variable appears in the current data segment (e.g., DSEG), converting the first form to assembly language is easy, just use the assembly language statement:

```
mov     variable, constant
```

This move immediate instruction copies the constant into the variable.

The second assignment above is somewhat complicated since the 80x86 doesn't provide a memory–to-memory mov instruction. Therefore, to copy one memory variable into another, you must move the data through a register. If you'll look at the encoding for the mov instruction in the appendix, you'll notice that the mov ax, memory and mov memory, ax instructions are shorter than moves involving other registers. Therefore, if the ax register is available, you should use it for this operation. For example,

```
var1 := var2;
```

becomes

```
mov     ax, var2
mov     var1, ax
```

Of course, if you're using the ax register for something else, one of the other registers will suffice. Regardless, you must use a register to transfer one memory location to another.

This discussion, of course, assumes that both variables are in memory. If possible, you should try to use a register to hold the value of a variable.

9.1.2 Simple Expressions

The next level of complexity up from a simple assignment is a simple expression. A simple expression takes the form:

```
var := term1 op term2;
```

Var is a variable, $term_1$ and $term_2$ are variables or constants, and op is some arithmetic operator (addition, subtraction, multiplication, etc.).

As simple as this expression appears, most expressions take this form. It should come as no surprise then, that the 80x86 architecture was optimized for just this type of expression.

A typical conversion for this type of expression takes the following form:

```
mov     ax, term1
op      ax, term2
mov     var, ax
```

Op is the mnemonic that corresponds to the specified operation (e.g., "+" = add, "-" = sub, etc.).

There are a few inconsistencies you need to be aware of. First, the 80x86's {i}mul instructions do not allow immediate operands on processors earlier than the 80286. Further, none of the processors allow immediate operands with {i}div. Therefore, if the operation is multiplication or division and one of the terms is a constant value, you may need to load this constant into a register or memory location and then multiply or divide ax by that value. Of course, when dealing with the multiply and divide instructions on the 8086/8088, you must use the ax and dx registers. You cannot use arbitrary registers as you can with other operations. Also, don't forget the sign extension instructions if you're performing a division operation and you're dividing one 16/32 bit number by another. Finally, don't forget that some instructions may cause overflow. You may want to check for an overflow (or underflow) condition after an arithmetic operation.

Examples of common simple expressions:

```
X := Y + Z;

                mov     ax, y
                add     ax, z
                mov     x, ax

X := Y - Z;

                mov     ax, y
                sub     ax, z
                mov     x, ax

X := Y * Z; {unsigned}

                mov     ax, y
                mul     z           ;Use IMUL for signed arithmetic.
                mov     x, ax       ;Don't forget this wipes out DX.

X := Y div Z; {unsigned div}

                mov     ax, y
                mov     dx, 0       ;Zero extend AX into DX
                div     z
                mov     x, ax

X := Y div Z; {signed div}

                mov     ax, y
                cwd                 ;Sign extend AX into DX
                idiv    z
                mov     x, ax

X := Y mod Z; {unsigned remainder}

                mov     ax, y
                mov     dx, 0       ;Zero extend AX into DX
                div     z
                mov     x, dx       ;Remainder is in DX
```

```
X := Y mod Z; {signed remainder}

                mov     ax, y
                cwd                     ;Sign extend AX into DX
                idiv    z
                mov     x, dx           ;Remainder is in DX
```

Since it is possible for an arithmetic error to occur, you should generally test the result of each expression for an error before or after completing the operation. For example, unsigned addition, subtraction, and multiplication set the carry flag if an overflow occurs. You can use the jc or jnc instructions immediately after the corresponding instruction sequence to test for overflow. Likewise, you can use the jo or jno instructions after these sequences to test for signed arithmetic overflow. The next two examples demonstrate how to do this for the add instruction:

```
X := Y + Z; {unsigned}

                mov     ax, y
                add     ax, z
                mov     x, ax
                jc      uOverflow

X := Y + Z; {signed}

                mov     ax, y
                add     ax, z
                mov     x, ax
                jo      sOverflow
```

Certain unary operations also qualify as simple expressions. A good example of a unary operation is negation. In a high level language negation takes one of two possible forms:

```
var := -var             or          var1 := -var2
```

Note that var := -constant is really a simple assignment, not a simple expression. You can specify a negative constant as an operand to the mov instruction:

```
                mov     var, -14
```

To handle the first form of the negation operation above use the single assembly language statement:

```
                neg     var
```

If two different variables are involved, then use the following:

```
                mov     ax, var2
                neg     ax
                mov     var1, ax
```

Overflow only occurs if you attempt to negate the most negative value (-128 for eight bit values, -32768 for sixteen bit values, etc.). In this instance the 80x86 sets the overflow flag, so you can test for arithmetic overflow using the jo or jno instructions. In all other cases the80x86 clears the overflow flag. The carry flag has no meaning after executing the neg instruction since neg (obviously) does not apply to unsigned operands.

9.1.3 Complex Expressions

A complex expression is any arithmetic expression involving more than two terms and one operator. Such expressions are commonly found in programs written in a high level language. Complex expressions may include parentheses to override operator precedence, function calls, array accesses, etc. While the conversion of some complex expressions to assembly language is fairly straight-forward, others require some effort. This section outlines the rules you use to convert such expressions.

A complex function that is easy to convert to assembly language is one that involves three terms and two operators, for example:

```
                W := W - Y - Z;
```

Clearly the straight-forward assembly language conversion of this statement will require two sub instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

```
        mov     ax, w
        sub     ax, y
        sub     ax, z
        mov     w, ax
```

and

```
        mov     ax, y
        sub     ax, z
        sub     w, ax
```

The second conversion, since it is shorter, looks better. However, it produces an incorrect result (assuming Pascal-like semantics for the original statement). Associativity is the problem. The second sequence above computes W := W - (Y - Z) which is not the same as W := (W - Y) - Z. How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

```
        mov     ax, y
        add     ax, z
        sub     w, ax
```

This computes W:=W-(Y+Z). This is equivalent to W := (W - Y) - Z.

Precedence is another issue. Consider the Pascal expression:

```
X := W * Y + Z;
```

Once again there are two ways we can evaluate this expression:

```
X := (W * Y) + Z;
```

or

```
X := W * (Y + Z);
```

By now, you're probably thinking that this text is crazy. Everyone knows the correct way to evaluate these expressions is the second form provided in these two examples. However, you're wrong to think that way. The APL programming language, for example, evaluates expressions solely from right to left and does not give one operator precedence over another.

Most high level languages use a fixed set of precedence rules to describe the order of evaluation in an expression involving two or more different operators. Most programming languages, for example, compute multiplication and division before addition and subtraction. Those that support exponentiation (e.g., FORTRAN and BASIC) usually compute that before multiplication and division. These rules are intuitive since almost everyone learns them before high school. Consider the expression:

$X\ op_1\ Y\ op_2\ Z$

If op_1 takes precedence over op_2 then this evaluates to (X op_1 Y) op_2 Z otherwise if op_2 takes precedence over op_1 then this evaluates to X op_1 (Y op_2 Z). Depending upon the operators and operands involved, these two computations could produce different results.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

```
; W := X + Y * Z;

        mov     bx, x
        mov     ax, y           ;Must compute Y * Z first since
        mul     z               ; "*" has the highest precedence.
        add     bx, ax          ;Now add product with X's value.
        mov     w, bx           ;Save away result.
```

Since addition is a *commutative* operation, we could optimize the above code to produce:

```
; W := X + Y * Z;

                mov     ax, y       ;Must compute Y * Z first since
                mul     z           ; "*" has the highest precedence.
                add     ax, x       ;Now add product with X's value.
                mov     w, ax       ;Save away result.
```

If two operators appearing within an expression have the same precedence, then you determine the order of evaluation using *associativity* rules. Most operators are *left associative* meaning that they evaluate from left to right. Addition, subtraction, multiplication, and division are all left associative. A *right associative* operator evaluates from right to left. The exponentiation operator in FORTRAN and BASIC is a good example of a right associative operator:

```
2^2^3 is equal to 2^(2^3) not (2^2)^3
```

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
; W := X - Y - Z

                mov     ax, x       ;All the same operator, so we need
                sub     ax, y       ; to evaluate from left to right
                sub     ax, z       ; because they all have the same
                mov     w, ax       ; precedence.

; W := X + Y * Z

                mov     ax, y       ;Must compute Y * Z first since
                imul    z           ; multiplication has a higher
                add     ax, x       ; precedence than addition.
                mov     w, ax

; W := X / Y - Z

                mov     ax, x       ;Here we need to compute division
                cwd                 ; first since it has the highest
                idiv    y           ; precedence.
                sub     ax, z
                mov     w, ax

; W := X * Y * Z

                mov     ax, y       ;Addition and multiplication are
                imul    z           ; commutative, therefore the order
                imul    x           ; of evaluation does not matter
                mov     w, ax
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is always better to perform the multiplication first. For example, given an expression of the form:

```
W := X/Y * Z
```

It is better to compute X*Z and then divide the result by Y rather than divide X by Y and multiply the quotient by Z. There are two reasons this approach is better. First, remember that the imul instruction always produces a 32 bit result (assuming 16 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the dx register so you do not have to sign extend ax prior to the division. This saves the execution of the cwd instruction. A second reason for doing the multiplication first is to increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute 5/2 you will get the value two, not 2.5. Computing (5/2)*3 produces six. However, if you compute (5*3)/2 you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

```
W := X/Y*Z;
```

You can usually convert this to assembly code:

```
mov     ax, x
imul    z
idiv    z
mov     w, ax
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following Pascal statement:

```
W := X - Y * Z;
```

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute Y * Z and then subtract X from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load X into a register, multiply Y and Z leaving their product in a different register, and then subtract this product from X, e.g.,

```
mov     bx, x
mov     ax, y
imul    z
sub     bx, ax
mov     w, bx
```

This is a trivial example that demonstrates the need for *temporary variables* in an expression. The code uses the bx register to temporarily hold a copy of X until it computes the product of Y and Z. As your expression increase in complexity, the need for temporaries grows. Consider the following Pascal statement:

```
W := (A + B) * (Y + Z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

```
Temp1 := A + B;
Temp2 := Y + Z;
W := Temp1 * Temp2;
```

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov     ax, a
add     ax, b
mov     Temp1, ax
mov     ax, y
add     ax, z
mov     temp2, ax
mov     ax, temp1,
imul    temp2
mov     w, ax
```

Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

```
mov     ax, a
add     ax, b
mov     bx, y
add     bx, z
imul    bx
mov     w, ax
```

Yet another example:

```
X := (Y+Z) * (A-B) / 10;
```

This can be converted to a set of four simple expressions:

```
Temp1 := (Y+Z)
Temp2 := (A-B)
Temp1 := Temp1 * Temp2
X := Temp1 / 10
```

You can convert these four simple expressions into the assembly language statements:

```
mov     ax, y       ;Compute AX := Y+Z
add     ax, z
mov     bx, a       ;Compute BX := A-B
sub     bx, b
mul     bx          ;Compute AX := AX * BX, this also sign
mov     bx, 10      ; extends AX into DX for idiv.
idiv    bx          ;Compute AX := AX / 10
mov     x, ax       ;Store result into X
```

The most important thing to keep in mind is that temporary values, if possible, should be kept in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the results. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

9.1.4 Commutative Operators

If "@" represents some operator, that operator is *commutative* if the following relationship is always true:

```
(A @ B) = (B @ A)
```

As you saw in the previous section, commutative operators are nice because the order of their operands is immaterial and this lets you rearrange a computation, often making that computation easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, you should always check to see if there is a better sequence you can use to improve the size or speed of your code. The following tables list the commutative and non-commutative operators you typically find in high level languages:

Table 46: Some Common Commutative Binary Operators

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& or &	Logical or bitwise AND
OR	|| or |	Logical or bitwise OR
XOR	^	(Logical or) Bitwise exclusive-OR
=	==	Equality
<>	!=	Inequality

Table 47: Some Common Noncommutative Binary Operators

Pascal	C/C++	Description
-	-	Subtraction
/ or DIV	/	Division
MOD	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

9.2 Logical (Boolean) Expressions

Consider the following expression from a Pascal program:

```
B := ((X=Y) and (A <= C)) or ((Z-A) <> 5);
```

B is a boolean variable and the remaining variables are all integers.

How do we represent boolean variables in assembly language? Although it takes only a single bit to represent a boolean value, most assembly language programmers allocate a whole byte or word for this purpose. With a byte, there are 256 possible values we can use to represent the two values *true* and *false*. So which two values (or which two sets of values) do we use to represent these boolean values? Because of the machine's architecture, it's much easier to test for conditions like zero or not zero and positive or negative rather than to test for one of two particular boolean values. Most programmers (and, indeed, some programming languages like "C") choose zero to represent false and anything else to represent true. Some people prefer to represent true and false with one and zero (respectively) and not allow any other values. Others select 0FFFFh for true and 0 for false. You could also use a positive value for true and a negative value for false. All these mechanisms have their own advantages and drawbacks.

Using only zero and one to represent false and true offers one very big advantage: the 80x86 logical instructions (and, or, xor and, to a lesser extent, not) operate on these values exactly as you would expect. That is, if you have two boolean variables A and B, then the following instructions perform the basic logical operations on these two variables:

```
mov     ax, A
and     ax, B
mov     C, ax       ;C := A and B;

mov     ax, A
or      ax, B
mov     C, ax       ;C := A or B;

mov     ax, A
xor     ax, B
mov     C, ax       ;C := A xor B;

mov     ax, A       ;Note that the NOT instruction does not
not     ax          ; properly compute B := not A by itself.
and     ax, 1       ; I.e., (NOT 0)does not equal one.
mov     B, ax       ;B := not A;

mov     ax, A       ;Another way to do B := NOT A;
xor     ax, 1
mov     B, ax       ;B := not A;
```

Note, as pointed out above, that the not instruction will not properly compute logical negation. The bitwise not of zero is 0FFh and the bitwise not of one is 0FEh. Neither result is zero or one. However, by anding the result with one you get the proper result. Note that

you can implement the not operation more efficiently using the xor ax, 1 instruction since it only affects the L.O. bit.

As it turns out, using zero for false and anything else for true has a lot of subtle advantages. Specifically, the test for true or false is often implicit in the execution of any logical instruction. However, this mechanism suffers from a very big disadvantage: you cannot use the 80x86 and, or, xor, and not instructions to implement the boolean operations of the same name. Consider the two values 55h and 0AAh. They're both non-zero so they both represent the value true. However, if you logically and 55h and 0AAh together using the 80x86 and instruction, the result is zero. (True and true) should produce true, not false. A system that uses non-zero values to represent true and zero to represent false is an *arithmetic logical system*. A system that uses the two distinct values like zero and one to represent false and true is called a *boolean logical system*, or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression:

```
B := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
```

The simple expressions resulting from this expression might be:

```
Temp2 := X = Y
Temp := A <= D
Temp := Temp and Temp2
Temp2 := Z-A
Temp2 := Temp2 <> 5
B := Temp or Temp2
```

The assembly language code for these expressions could be:

```
                mov     ax, x       ;See if X = Y and load zero or
                cmp     ax, y       ; one into AX to denote the result
                jnz     L1          ; of this comparison.
                mov     al, 1       ;X = Y
                jmp     L2
L1:             mov     al, 0       ;X <> Y
L2:
                mov     bx, A       ;See if A <= D and load zero or one
                cmp     bx, D       ; into BX to denote the result of
                jle     ST1         ; this comparison.
                mov     bl, 0
                jmp     L3
ST1:            mov     bl, 1
L3:
                and     bl, al      ;Temp := Temp and Temp2

                mov     ax, Z       ;See if (Z-A) <> 5.
                sub     ax, A       ;Temp2 := Z-A;
                cmp     ax, 5       ;Temp2 := Temp2 <> 5;
                jnz     ST2
                mov     al, 0
                jmp     short L4
ST2:            mov     al, 1
L4:
                or      al, bl      ;Temp := Temp or Temp2;
                mov     B, al       ;B := Temp;
```

As you can see, this is a rather unwieldy sequence of statements. One slight optimization you can use is to assume a result is going to be true or false and initialize the corresponding boolean result ahead of time:

```
                mov     bl, 0       ;Assume X <> Y
                mov     ax, x
                cmp     ax, Y
                jne     L1
                mov     bl, 1       ;X is equal to Y, so make this true.
L1:
                mov     bh, 0       ;Assume not (A <= D)
                mov     ax, A
                cmp     ax, D
                jnle    L2
                mov     bh, 1       ;A <= D so make this true
```

```
L2:
                and     bl, bh      ;Compute logical AND of results.

                mov     bh, 0       ;Assume (Z-A) = 5
                mov     ax, Z
                sub     ax, A
                cmp     ax, 5
                je      L3:
                mov     bh, 1       ;(Z-A) <> 5
L3:
                or      bl, bh      ;Logical OR of results.
                mov     B, bl       ;Save boolean result.
```

Of course, if you have an 80386 or later processor, you can use the set*cc* instructions to simplify this a bit:

```
                mov     ax, x
                cmp     ax, y
                sete    al          ;TEMP2 := X = Y

                mov     bx, A
                cmp     bx, D
                setle   bl          ;TEMP := A <= D
                and     bl, al      ;Temp := Temp and Temp2
                mov     ax, Z
                sub     ax, A       ;Temp2 := Z-A;
                cmp     ax, 5       ;Temp2 := Temp2 <> 5;
                setne   al
                or      bl, al      ;Temp := Temp or Temp2;
                mov     B, bl       ;B := Temp;
```

This code sequence is obviously much better than the previous one, but it will only execute on 80386 and later processors.

Another way to handle boolean expressions is to represent boolean values by states within your code. The basic idea is to forget maintaining a boolean variable throughout the execution of a code sequence and use the position within the code to determine the boolean result. Consider the following implementation of the above expression. First, let's rearrange the expression to be

```
B := ((Z-A) <> 5) or ((X=Y) and (A <= D));
```

This is perfectly legal since the or operation is commutative. Now consider the following implementation:

```
                mov     B, 1        ;Assume the result is true.
                mov     ax, Z       ;See if (Z-A) <> 5
                sub     ax, A       ;If this condition is true, the
                cmp     ax, 5       ; result is always true so there
                jne     Done        ; is no need to check the rest.

                mov     ax, X       ;If X <> Y, the result is false,
                cmp     ax, Y       ; no matter what A and D contain
                jne     SetBtoFalse

                mov     ax, A       ;Now see if A <= D.
                cmp     ax, D
                jle     Done        ;If so, quit.
SetBtoFalse:    mov     B, 0        ;If B is false, handle that here.
Done:
```

Notice that this section of code is a lot shorter than the first version above (and it runs on all processors). The previous translations did everything computationally. This version uses program flow logic to improve the code. It begins by assuming a true result and sets the B variable to true. It then checks to see if (Z-A) <> 5. If this is true the code branches to the done table because B is true no matter what else happens. If the program falls through to the mov ax, X instruction, we know that the result of the previous comparison is false. There is no need to save this result in a temporary since we implicitly know its value by the fact that we're executing the mov ax, X instruction. Likewise, the second group of statements above checks to see if X is equal to Y. If it is not, we already know the result is false

so this code jumps to the SetBtoFalse label. If the program begins executing the third set of statements above, we know that the first result was false and the second result was true; the position of the code guarantees this. Therefore, there is no need to maintain temporary boolean variables that keep track of the state of this computation.

Consider another example:

```
B := ((A = E) or (F <> D)) and ((A<>B) or (F = D));
```

Computationally, this expression would yield a considerable amount of code. However, by using flow control you can reduce it to the following:

```
                mov     b, 0        ;Assume result is false.
                mov     ax, a       ;See if A = E.
                cmp     ax, e
                je      test2       ;If so, 1st subexpression is true.

                mov     ax, f       ;If not, check 2nd subexpression
                cmp     ax, d       ; to see if F <> D.
                je      Done        ;If so, we're done, else fall
                                    ; through to next tests.
Test2:          mov     ax, a       ;Does A <> B?
                cmp     ax, b
                jne     SetBto1     ;If so, we're done.

                mov     ax, f       ;If not, see if F = D.
                cmp     ax, d
                jne     Done

SetBto1:        mov     b, 1
Done:
```

There is one other difference between using control flow vs. computation logic: when using control flow methods, you may skip the majority of the instructions that implement the boolean formula. This is known as *short-circuit evaluation*. When using the computation model, even with the set*cc* instruction, you wind up executing most of the statements. Keep in mind that this is not necessarily a disadvantage. On pipelined processors it may be much faster to execute several additional instructions rather than flush the pipeline and prefetch queue. You may need to experiment with your code to determine the best solution.

When working with boolean expressions don't forget the that you might be able to optimize your code by simplifying those boolean expressions (see "Simplification of Boolean Functions" on page 52). You can use algebraic transformations (especially DeMorgan's theorems) and the mapping method to help reduce the complexity of an expression.

9.3 Multiprecision Operations

One big advantage of assembly language over HLLs is that assembly language does not limit the size of integers. For example, the C programming language defines a maximum of three different integer sizes: short int, int, and long int. On the PC, these are often 16 or 32 bit integers. Although the 80x86 machine instructions limit you to processing eight, sixteen, or thirty-two bit integers with a single instruction, you can always use more than one instruction to process integers of any size you desire. If you want 256 bit integer values, no problem. The following sections describe how extended various arithmetic and logical operations from 16 or 32 bits to as many bits as you please.

9.3.1 Multiprecision Addition Operations

The 80x86 add instruction adds two 8, 16, or 32 bit numbers[1]. After the execution of the add instruction, the 80x86 carry flag is set if there is an overflow out of the H.O. bit of

1. As usual, 32 bit arithmetic is available only on the 80386 and later processors.

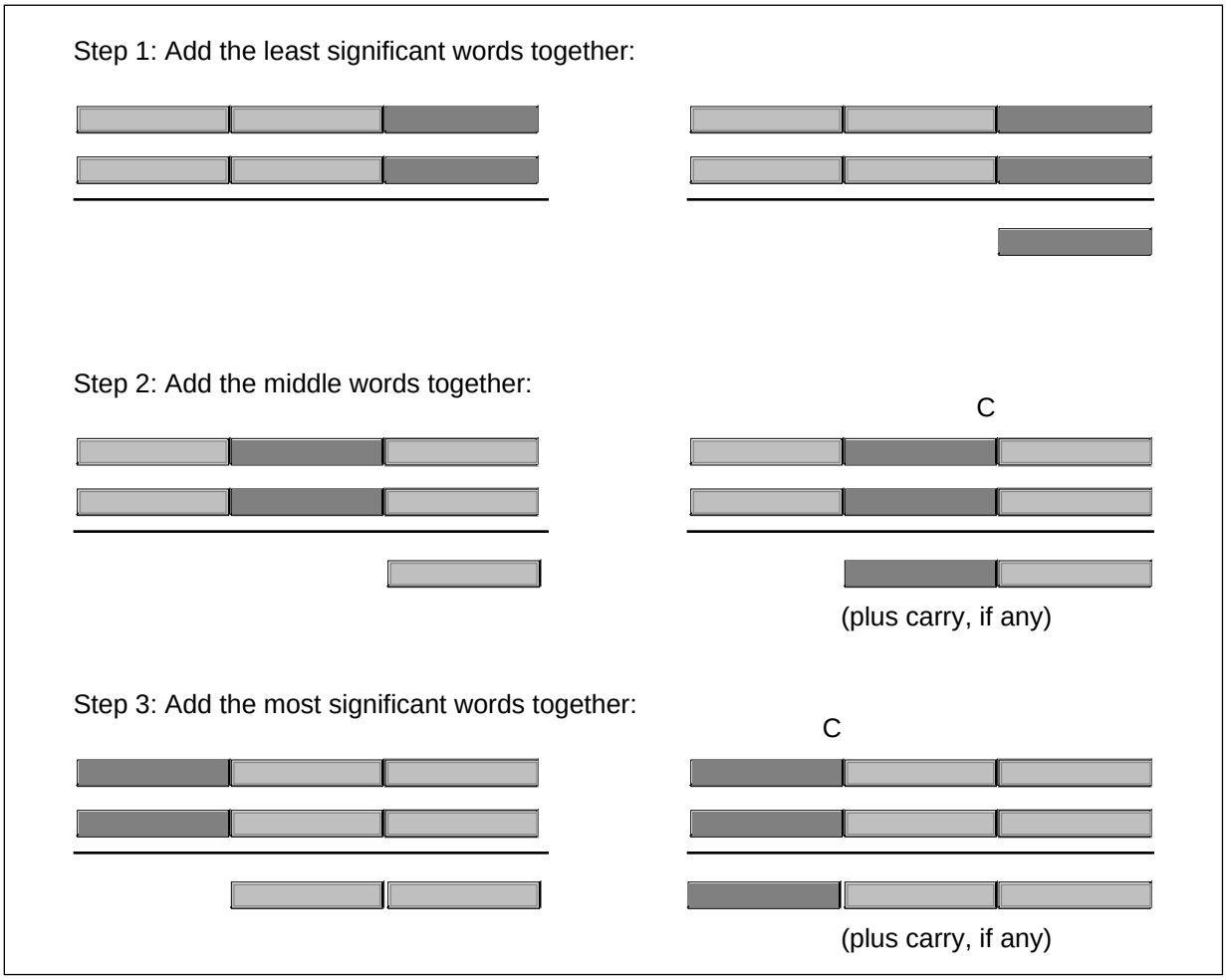


Figure 8.1 Multiprecision (48-bit) Addition

the sum. You can use this information to do multiprecision addition operations. Consider the way you manually perform a multidigit (multiprecision) addition operation:

Step 1: Add the least significant digits together:

```
 289                              289
+456          produces           +456
----                             ----
                                    5 with carry 1.
```

Step 2: Add the next significant digits plus the carry:

```
  1 (previous carry)
 289                                    289
+456                  produces         +456
----                                   ----
   5                                     45 with carry 1.
```

Step 3: Add the most significant digits plus the carry:

```
                                       1 (previous carry)
 289                                    289
+456                  produces         +456
----                                   ----
  45                                    745
```

The 80x86 handles extended precision arithmetic in an identical fashion, except instead of adding the numbers a digit at a time, it adds them a byte or a word at a time. Consider the three-word (48 bit) addition operation in Figure 8.1.

The add instruction adds the L.O. words together. The adc (add with carry) instruction adds all other word pairs together. The adc instruction adds two operands plus the carry flag together producing a word value and (possibly) a carry.

For example, suppose that you have two thirty-two bit values you wish to add together, defined as follows:

```
X               dword       ?
Y               dword       ?
```

Suppose, also, that you want to store the sum in a third variable, Z, that is likewise defined with the dword directive. The following 80x86 code will accomplish this task:

```
                mov     ax, word ptr X
                add     ax, word ptr Y
                mov     word ptr Z, ax
                mov     ax, word ptr X+2
                adc     ax, word ptr Y+2
                mov     word ptr Z+2, ax
```

Remember, these variables are declared with the dword directive. Therefore the assembler will not accept an instruction of the form mov ax, X because this instruction would attempt to load a 32 bit value into a 16 bit register. Therefore this code uses the *word ptr* coercion operator to coerce symbols X, Y, and Z to 16 bits. The first three instructions add the L.O. words of X and Y together and store the result at the L.O. word of Z. The last three instructions add the H.O. words of X and Y together, along with the carry out of the L.O. word, and store the result in the H.O. word of Z. Remember, address expressions of the form "X+2" access the H.O. word of a 32 bit entity. This is due to the fact that the 80x86 address space addresses bytes and it takes two consecutive bytes to form a word.

Of course, if you have an 80386 or later processor you needn't go through all this just to add two 32 bit values together, since the 80386 directly supports 32 bit operations. However, if you wanted to add two *64 bit* integers together on the 80386, you would still need to use this technique.

You can extend this to any number of bits by using the adc instruction to add in the higher order words in the values. For example, to add together two 128 bit values, you could use code that looks something like the following:

```
BigVal1         dword   0,0,0,0         ;Four double words in 128 bits!
BigVal2         dword   0,0,0,0
BigVal3         dword   0,0,0,0
                  .
                  .
                  .
                mov     eax, BigVal1    ;No need for dword ptr operator since
                add     eax, BigVal2    ; these are dword variables.
                mov     BigVal3, eax

                mov     eax, BigVal1+4  ;Add in the values from the L.O.
                adc     eax, BigVal2+4  ; entity to the H.O. entity using
                mov     BigVal3+4, eax  ; the ADC instruction.

                mov     eax, BigVal1+8
                adc     eax, BigVal2+8
                mov     BigVal3+8, eax

                mov     eax, BigVal1+12
                adc     eax, BigVal2+12
                mov     BigVal3+12, eax
```

9.3.2 Multiprecision Subtraction Operations

Like addition, the 80x86 performs multi-byte subtraction the same way you would manually, except it subtracts whole bytes , words, or double words at a time rather than decimal digits. The mechanism is similar to that for the add operation, You use the sub instruction on the L.O. byte / word / double word and the sbb instruction on the high order

values. The following example demonstrates a 32 bit subtraction using the 16 bit registers on the 8086:

```
var1            dword       ?
var2            dword       ?
diff            dword       ?

                mov         ax, word ptr var1
                sub         ax, word ptr var2
                mov         word ptr diff, ax
                mov         ax, word ptr var1+2
                sbb         ax, word ptr var2+2
                mov         word ptr diff+2, ax
```

The following example demonstrates a 128-bit subtraction using the 80386 32 bit register set:

```
BigVal1         dword       0,0,0,0         ;Four double words in 128 bits!
BigVal2         dword       0,0,0,0
BigVal3         dword       0,0,0,0
                  .
                  .
                  .
                mov         eax, BigVal1    ;No need for dword ptr operator since
                sub         eax, BigVal2    ; these are dword variables.
                mov         BigVal3, eax

                mov         eax, BigVal1+4  ;Subtract the values from the L.O.
                sbb         eax, BigVal2+4  ; entity to the H.O. entity using
                mov         BigVal3+4, eax  ; the SUB and SBB instructions.

                mov         eax, BigVal1+8
                sbb         eax, BigVal2+8
                mov         BigVal3+8, eax

                mov         eax, BigVal1+12
                sbb         eax, BigVal2+12
                mov         BigVal3+12, eax
```

9.3.3 Extended Precision Comparisons

Unfortunately, there isn't a "compare with borrow" instruction that can be used to perform extended precision comparisons. Since the cmp and sub instructions perform the same operation, at least as far as the flags are concerned, you'd probably guess that you could use the sbb instruction to synthesize an extended precision comparison; however, you'd only be partly right. There is, however, a better way.

Consider the two unsigned values 2157h and 1293h. The L.O. bytes of these two values do not affect the outcome of the comparison. Simply comparing 21h with 12h tells us that the first value is greater than the second. In fact, the only time you ever need to look at both bytes of these values is if the H.O. bytes are equal. In all other cases comparing the H.O. bytes tells you everything you need to know about the values. Of course, this is true for any number of bytes, not just two. The following code compares two signed 64 bit integers on an 80386 or later processor:

```
; This sequence transfers control to location "IsGreater" if
; QwordValue > QwordValue2. It transfers control to "IsLess" if
; QwordValue < QwordValue2. It falls though to the instruction
; following this sequence if QwordValue = QwordValue2. To test for
; inequality, change the "IsGreater" and "IsLess" operands to "NotEqual"
; in this code.

                mov     eax, dword ptr QWordValue+4    ;Get H.O. dword
                cmp     eax, dword ptr QWordValue2+4
                jg      IsGreater
                jl      IsLess
                mov     eax, dword ptr QWordValue
                cmp     eax, dword ptr QWordValue2
                jg      IsGreater
                jl      IsLess
```

To compare unsigned values, simply use the ja and jb instructions in place of jg and jl.

You can easily synthesize any possible comparison from the sequence above, the following examples show how to do this. These examples do signed comparisons, substitute ja, jae, jb, and jbe for jg, jge, jl, and jle (respectively) to do unsigned comparisons.

```
QW1             qword   ?
QW2             qword   ?

dp              textequ <dword ptr>

; 64 bit test to see if QW1 < QW2 (signed).
; Control transfers to "IsLess" label if QW1 < QW2. Control falls
; through to the next statement if this is not true.

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jg      NotLess
                jl      IsLess
                mov     eax, dp QW1             ;Fall through to here if H.O.
                cmp     eax, dp QW2             ; dwords are equal.
                jl      IsLess
NotLess:

; 64 bit test to see if QW1 <= QW2 (signed).

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jg      NotLessEq
                jl      IsLessEq
                mov     eax, dp QW1
                cmp     eax, dword ptr QW2
                jle     IsLessEq
NotLessEQ:

; 64 bit test to see if QW1 >QW2 (signed).

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jg      IsGtr
                jl      NotGtr
                mov     eax, dp QW1             ;Fall through to here if H.O.
                cmp     eax, dp QW2             ; dwords are equal.
                jg      IsGtr
NotGtr:

; 64 bit test to see if QW1 >= QW2 (signed).

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jg      IsGtrEq
                jl      NotGtrEq
                mov     eax, dp QW1
                cmp     eax, dword ptr QW2
                jge     IsGtrEq
NotGtrEq:

; 64 bit test to see if QW1 = QW2 (signed or unsigned). This code branches
; to the label "IsEqual" if QW1 = QW2. It falls through to the next instruction
; if they are not equal.

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jne     NotEqual
                mov     eax, dp QW1
                cmp     eax, dword ptr QW2
                je      IsEqual
NotEqual:
```

```
; 64 bit test to see if QW1 <> QW2 (signed or unsigned). This code branches
; to the label "NotEqual" if QW1 <> QW2. It falls through to the next
; instruction if they are equal.

                mov     eax, dp QW1+4           ;Get H.O. dword
                cmp     eax, dp QW2+4
                jne     NotEqual
                mov     eax, dp QW1
                cmp     eax, dword ptr QW2
                jne     NotEqual
```

9.3.4 Extended Precision Multiplication

Although a 16x16 or 32x32 multiply is usually sufficient, there are times when you may want to multiply larger values together. You will use the 80x86 single operand mul and imul instructions for extended precision multiplication.

Not surprisingly (in view of how adc and sbb work), you use the same techniques to perform extended precision multiplication on the 80x86 that you employ when manually multiplying two values.

Consider a simplified form of the way you perform multi-digit multiplication by hand:

```
1) Multiply the first two
   digits together (5*3):

      123
       45
      ---
       15


2) Multiply 5*2:

      123
       45
       ---
       15
      10


3) Multiply 5*1:

      123
       45
      ---
       15
      10
      5


4) 4*3:

      123
       45
       ---
       15
      10
      5
      12


5) Multiply 4*2:

       123
        45
       ---
        15
       10
       5
       12
       8


6) 4*1:

        123
         45
        ---
         15
        10
        5
        12
        8
       4


7) Add all the partial products together:

       123
        45
       ---
        15
       10
       5
       12
       8
      4
      ------
      5535
```

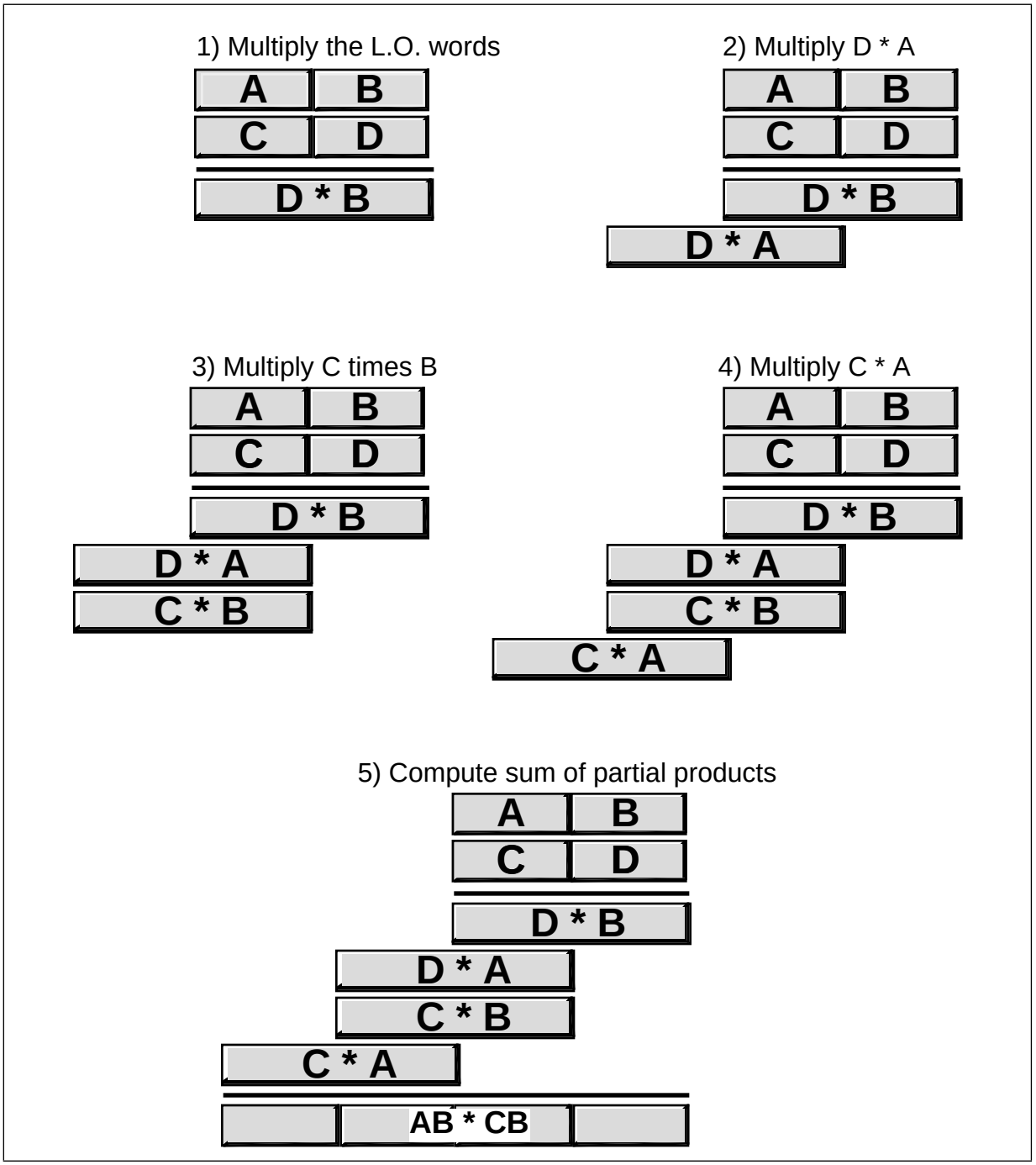


Figure 8.2 Multiprecision Multiplication

The 80x86 does extended precision multiplication in the same manner except that it works with bytes, words, and double words rather than digits. Figure 8.2 shows how this works.

Probably the most important thing to remember when performing an extended precision multiplication is that you must also perform a multiple precision addition at the same time. Adding up all the partial products requires several additions that will produce the result. The following listing demonstrates the proper way to multiply two 32 bit values on a 16 bit processor:

Note: Multiplier and Multiplicand are 32 bit variables declared in the data segment via the dword directive. Product is a 64 bit variable declared in the data segment via the qword directive.

```
Multiply        proc    near
                push    ax
                push    dx
                push    cx
                push    bx

; Multiply the L.O. word of Multiplier times Multiplicand:

                mov     ax, word ptr Multiplier
                mov     bx, ax                      ;Save Multiplier val
                mul     word ptr Multiplicand       ;Multiply L.O. words
                mov     word ptr Product, ax        ;Save partial product
                mov     cx, dx                      ;Save H.O. word

                mov     ax, bx                      ;Get Multiplier in BX
                mul     word ptr Multiplicand+2     ;Multiply L.O. * H.O.
                add     ax, cx                      ;Add partial product
                adc     dx, 0                       ;Don't forget carry!
                mov     bx, ax                      ;Save partial product
                mov     cx, dx                      ; for now.

; Multiply the H.O. word of Multiplier times Multiplicand:

                mov     ax, word ptr Multiplier+2   ;Get H.O. Multiplier
                mul     word ptr Multiplicand       ;Times L.O. word
                add     ax, bx                      ;Add partial product
                mov     word ptr product+2, ax      ;Save partial product
                adc     cx, dx                      ;Add in carry/H.O.!

                mov     ax, word ptr Multiplier+2   ;Multiply the H.O.
                mul     word ptr Multiplicand+2     ; words together.
                add     ax, cx                      ;Add partial product
                adc     dx, 0                       ;Don't forget carry!
                mov     word ptr Product+4, ax      ;Save partial product
                mov     word ptr Product+6, dx

                pop     bx
                pop     cx
                pop     dx
                pop     ax
                ret
Multiply        endp
```

One thing you must keep in mind concerning this code, it only works for unsigned operands. Multiplication of signed operands appears in the exercises.

9.3.5 Extended Precision Division

You cannot synthesize a general n-bit / m-bit division operation using the div and idiv instructions. Such an operation must be performed using a sequence of shift and subtract instructions. Such an operation is extremely messy. A less general operation, dividing an n bit quantity by a 32 bit (on the 80386 or later) or 16 bit quantity is easily synthesized using the div instruction. The following code demonstrates how to divide a 64 bit quantity by a 16 bit divisor, producing a 64 bit quotient and a 16 bit remainder:

```
dseg            segment  para public 'DATA'
dividend        dword    0FFFFFFFFh, 12345678h
divisor         word     16
Quotient        dword    0,0
Modulo          word     0
dseg            ends

cseg            segment  para public 'CODE'
                assume   cs:cseg, ds:dseg

; Divide a 64 bit quantity by a 16 bit quantity:

Divide64        proc     near

                mov      ax, word ptr dividend+6
                sub      dx, dx
```

```
                div     divisor
                mov     word ptr Quotient+6, ax
                mov     ax, word ptr dividend+4
                div     divisor
                mov     word ptr Quotient+4, ax
                mov     ax, word ptr dividend+2
                div     divisor
                mov     word ptr Quotient+2, ax
                mov     ax, word ptr dividend
                div     divisor
                mov     word ptr Quotient, ax
                mov     Modulo, dx
                ret
Divide64        endp
cseg            ends
```

This code can be extended to any number of bits by simply adding additional mov / div / mov instructions at the beginning of the sequence. Of course, on the 80386 and later processors you can divide by a 32 bit value by using edx and eax in the above sequence (with a few other appropriate adjustments).

If you need to use a divisor larger than 16 bits (32 bits on an 80386 or later), you're going to have to implement the division using a shift and subtract strategy. Unfortunately, such algorithms are very slow. In this section we'll develop two division algorithms that operate on an arbitrary number of bits. The first is slow but easier to understand, the second is quite a bit faster (in general).

As for multiplication, the best way to understand how the computer performs division is to study how you were taught to perform long division by hand. Consider the operation 3456/12 and the steps you would take to manually perform this operation:

```
12 |3456
   24
```

(1) 12 goes into 34 two times.

```
    2
12 |3456
   24
   105
```

(2) Subtract 24 from 35 and drop down the 105.

```
    2
12 |3456
   24
   105
    96
```

(3) 12 goes into 105 eight times.

```
    28
12 |3456
   24
   105
    96
     96
```

(4) Subtract 96 from 105 and drop down the 96.

```
    28
12 |3456
   24
   105
    96
     96
     96
```

(5) 12 goes into 96 exactly eight times.

```
    288
12 |3456
   24
   105
    96
     96
     96
```

(6) Therefore, 12 goes into 3456 exactly 288 times.

This algorithm is actually easier in binary since at each step you do not have to guess how many times 12 goes into the remainder nor do you have to multiply 12 by your guess to obtain the amount to subtract. At each step in the binary algorithm the divisor goes into the remainder exactly zero or one times. As an example, consider the division of 27 (11011) by three (11):

```
11 |11011
    11
```

11 goes into 11 one time.

```
     1
11 |11011
    11
     00
```

Subtract out the 11 and bring down the zero.

```
     1
11 |11011
    11
     00
     00
```

11 goes into 00 zero times.

```
    10
11 |11011
    11
     00
     00
      01
```

Subtract out the zero and bring down the one.

```
    10
11 |11011
    11
     00
     00
      01
      00
```

11 goes into 01 zero times.

```
    100
11 |11011
    11
     00
     00
      01
      00
       11
```

Subtract out the zero and bring down the one.

```
    100
11 |11011
    11
     00
     00
      01
      00
       11
       11
```

11 goes into 11 one time.

```
    1001
11 |11011
    11
     00
     00
      01
      00
       11
       11
       00
```

This produces the final result of 1001.

There is a novel way to implement this binary division algorithm that computes the quotient and the remainder at the same time. The algorithm is the following:

```
Quotient := Dividend;
Remainder := 0;
for i:= 1 to NumberBits do

       Remainder:Quotient := Remainder:Quotient SHL 1;
       if Remainder >= Divisor then

                Remainder := Remainder - Divisor;
                Quotient := Quotient + 1;

       endif
endfor
```

NumberBits is the number of bits in the Remainder, Quotient, Divisor, and Dividend variables. Note that the Quotient := Quotient + 1 statement sets the L.O. bit of Quotient to one since this algorithm previously shifts Quotient one bit to the left. The 80x86 code to implement this algorithm is

```
; Assume Dividend (and Quotient) is DX:AX, Divisor is in CX:BX,
; and Remainder is in SI:DI.

                mov     bp, 32      ;Count off 32 bits in BP
                sub     si, si      ;Set remainder to zero
                sub     di, di
BitLoop:        shl     ax, 1       ;See the section on shifts
                rcl     dx, 1       ; that describes how this
                rcl     di, 1       ; 64 bit SHL operation works
                rcl     si, 1
                cmp     si, cx      ;Compare H.O. words of Rem,
                ja      GoesInto    ; Divisor.
                jb      TryNext
                cmp     di, bx      ;Compare L.O. words.
                jb      TryNext

GoesInto:       sub     di, bx      ;Remainder := Remainder -
                sbb     si, cx      ;                Divisor
                inc     ax          ;Set L.O. bit of AX
TryNext:        dec     bp          ;Repeat 32 times.
                jne     BitLoop
```

This code looks short and simple, but there are a few problems with it. First, it does not check for division by zero (it will produce the value 0FFFFFFFFh if you attempt to divide by zero), it only handles unsigned values, and it is very slow. Handling division by zero is very simple, just check the divisor against zero prior to running this code and return an appropriate error code if the divisor is zero. Dealing with signed values is equally simple, you'll see how to do that in a little bit. The performance of this algorithm, however, leaves a lot to be desired. Assuming one pass through the loop takes about 30 clock cycles[2], this algorithm would require almost 1,000 clock cycles to complete! That's an order of magnitude worse than the DIV / IDIV instructions on the 80x86 that are among the slowest instructions on the 80x86.

There is a technique you can use to boost the performance of this division by a fair amount: check to see if the divisor variable really uses 32 bits. Often, even though the divisor is a 32 bit variable, the value itself fits just fine into 16 bits (i.e., the H.O. word of Divisor is zero). In this special case, that occurs frequently, you can use the DIV instruction which is much faster.

9.3.6 Extended Precision NEG Operations

Although there are several ways to negate an extended precision value, the shortest way is to use a combination of neg and sbb instructions. This technique uses the fact that neg subtracts its operand from zero. In particular, it sets the flags the same way the sub

2. This will vary depending upon your choice of processor.

instruction would if you subtracted the destination value from zero. This code takes the following form:

```
neg     dx
neg     ax
sbb     dx,0
```

The sbb instruction decrements dx if there is a borrow out of the L.O. word of the negation operation (which always occurs unless ax is zero).

To extend this operation to additional bytes, words, or double words is easy; all you have to do is start with the H.O. memory location of the object you want to negate and work towards the L.O. byte. The following code computes a 128 bit negation on the 80386 processor:

```
Value           dword   0,0,0,0     ;128 bit integer.
                 .
                 .
                 .
                neg     Value+12    ;Neg H.O. dword
                neg     Value+8     ;Neg previous dword in memory.
                sbb     Value+12, 0 ;Adjust H.O. dword
                neg     Value+4     ;Neg the second dword in object.
                sbb     Value+8, 0  ;Adjust 3rd dword in object.
                sbb     Value+12, 0 ;Carry any borrow through H.O. word.
                neg     Value       ;Negate L.O. word.
                sbb     Value+4, 0  ;Adjust 2nd dword in object.
                sbb     Value+8, 0  ;Adjust 3rd dword in object.
                sbb     Value+12, 0 ;Carry any borrow through H.O. word.
```

Unfortunately, this code tends to get really large and slow since you need to propogate the carry through all the H.O. words after each negate operation. A simpler way to negate larger values is to simply subract that value from zero:

```
Value           dword   0,0,0,0,0   ;160 bit integer.
                 .
                 .
                 .
                mov     eax, 0
                sub     eax, Value
                mov     Value, eax
                mov     eax, 0
                sbb     eax, Value+4
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+8
                mov     Value+8, ax
                mov     eax, 0
                sbb     eax, Value+12
                mov     Value+12, ax
                mov     eax, 0
                sbb     eax, Value+16
                mov     Value+16, ax
```

9.3.7 Extended Precision AND Operations

Performing an n-word and operation is very easy – simply and the corresponding words between the two operands, saving the result. For example, to perform the and operation where all three operands are 32 bits long, you could use the following code:

```
mov     ax, word ptr source1
and     ax, word ptr source2
mov     word ptr dest, ax
mov     ax, word ptr source1+2
and     ax, word ptr source2+2
mov     word ptr dest+2, ax
```

This technique easily extends to any number of words, all you need to is logically and the corresponding bytes, words, or double words in the corresponding operands.

9.3.8 Extended Precision OR Operations

Multi-word logical or operations are performed in the same way as multi-word and operations. You simply or the corresponding words in the two operand together. For example, to logically or two 48 bit values, use the following code:

```
mov     ax, word ptr operand1
or      ax, word ptr operand2
mov     word ptr operand3, ax
mov     ax, word ptr operand1+2
or      ax, word ptr operand2+2
mov     word ptr operand3+2, ax
mov     ax, word ptr operand1+4
or      ax, word ptr operand2+4
mov     word ptr operand3+4, ax
```

9.3.9 Extended Precision XOR Operations

Extended precision xor operations are performed in a manner identical to and/or – simply xor the corresponding words in the two operands to obtain the extended precision result. The following code sequence operates on two 64 bit operands, computes their exclusive-or, and stores the result into a 64 bit variable. This example uses the 32 bit registers available on 80386 and later processors.

```
mov     eax, dword ptr operand1
xor     eax, dword ptr operand2
mov     dword ptr operand3, eax
mov     eax, dword ptr operand1+4
xor     eax, dword ptr operand2+4
mov     dword ptr operand3+4, eax
```

9.3.10 Extended Precision NOT Operations

The not instruction inverts all the bits in the specified operand. It does not affect any flags (therefore, using a conditional jump after a not instruction has no meaning). An extended precision not is performed by simply executing the not instruction on all the affected operands. For example, to perform a 32 bit not operation on the value in (dx:ax), all you need to do is execute the instructions:

```
not     ax          or                  not     dx
not     dx                              not     ax
```

Keep in mind that if you execute the not instruction twice, you wind up with the original value. Also note that exclusive-oring a value with all ones (0FFh, 0FFFFh, or 0FF..FFh) performs the same operation as the not instruction.

9.3.11 Extended Precision Shift Operations

Extended precision shift operations require a shift and a rotate instruction. Consider what must happen to implement a 32 bit shl using 16 bit operations:

1) A zero must be shifted into bit zero.

2) Bits zero through 14 are shifted into the next higher bit.

3) Bit 15 is shifted into bit 16.

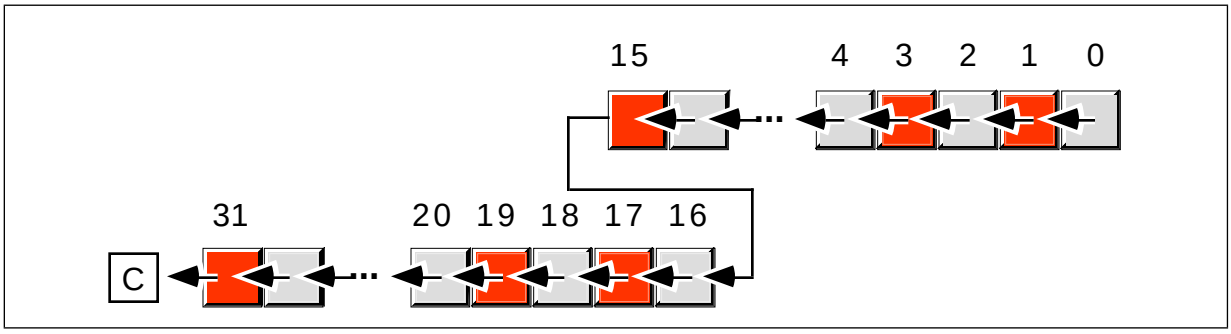


Figure 8.3 32-bit Shift Left Operation

4) Bits 16 through 30 must be shifted into the next higher bit.

5) Bit 31 is shifted into the carry flag.

The two instructions you can use to implement this 32 bit shift are shl and rcl. For example, to shift the 32 bit quantity in (dx:ax) one position to the left, you'd use the instructions:

```
shl     ax, 1
rcl     dx, 1
```

Note that you can only shift an extended precision value one bit at a time. You cannot shift an extended precision operand several bits using the cl register or an immediate value greater than one as the count using this technique

To understand how this instruction sequence works, consider the operation of these instructions on an individual basis. The shl instruction shifts a zero into bit zero of the 32 bit operand and shifts bit 15 into the carry flag. The rcl instruction then shifts the carry flag into bit 16 and then shifts bit 31 into the carry flag. The result is exactly what we want.

To perform a shift left on an operand larger than 32 bits you simply add additional rcl instructions. An extended precision shift left operation always starts with the least significant word and each succeeding rcl instruction operates on the next most significant word. For example, to perform a 48 bit shift left operation on a memory location you could use the following instructions:

```
shl     word ptr Operand, 1
rcl     word ptr Operand+2, 1
rcl     word ptr Operand+4, 1
```

If you need to shift your data by two or more bits, you can either repeat the above sequence the desired number of times (for a constant number of shifts) or you can place the instructions in a loop to repeat them some number of times. For example, the following code shifts the 48 bit value Operand to the left the number of bits specified in cx:

```
ShiftLoop:      shl     word ptr Operand, 1
                rcl     word ptr Operand+2, 1
                rcl     word ptr Operand+4, 1
                loop    ShiftLoop
```

You implement shr and sar in a similar way, except you must start at the H.O. word of the operand and work your way down to the L.O. word:

```
DblSAR:         sar word ptr Operand+4, 1
                rcr word ptr Operand+2, 1
                rcr word ptr Operand, 1

DblSHR:         shr word ptr Operand+4, 1
                rcr word ptr Operand+2, 1
                rcr word ptr Operand, 1
```

There is one major difference between the extended precision shifts described here and their 8/16 bit counterparts – the extended precision shifts set the flags differently than

the single precision operations. For example, the zero flag is set if the last rotate instruction produced a zero result, not if the entire shift operation produced a zero result. For the shift right operations, the overflow, and sign flags aren't set properly (they are set properly for the left shift operation). Additional testing will be required if you need to test one of these flags after an extended precision shift operation. Fortunately, the carry flag is the flag most often tested after a shift operation and the extended precision shift instructions properly set this flag.

The shld and shrd instructions let you efficiently implement multiprecision shifts of several bits on 80386 and later processors. Consider the following code sequence:

```
ShiftMe         dword   1234h, 5678h, 9012h
                 .
                 .
                 .
                mov     eax, ShiftMe+4
                shld    ShiftMe+8, eax, 6
                mov     eax, ShiftMe
                shld    ShiftMe+4, eax, 6
                shl     ShiftMe, 6
```

Recall that the shld instruction shifts bits from its second operand into its first operand. Therefore, the first shld instruction above shifts the bits from ShiftMe+4 into ShiftMe+8 *without affecting the value in ShiftMe+4*. The second shld instruction shifts the bits from ShiftMe into ShiftMe+4. Finally, the shl instruction shifts the L.O. double word the appropriate amount. There are two important things to note about this code. First, unlike the other extended precision shift left operations, this sequence works from the H.O. double word down to the L.O. double word. Second, the carry flag does not contain the carry out of the H.O. shift operation. If you need to preserve the carry flag at that point, you will need to push the flags after the first shld instruction and pop the flags after the shl instruction.

You can do an extended precision shift right operation using the shrd instruction. It works almost the same way as the code sequence above except you work from the L.O. double word to the H.O. double word. The solution is left as an exercise at the end of this chapter.

9.3.12 Extended Precision Rotate Operations

The rcl and rcr operations extend in a manner almost identical to that for shl and shr . For example, to perform 48 bit rcl and rcr operations, use the following instructions:

```
                rcl     word ptr operand,1
                rcl     word ptr operand+2, 1
                rcl     word ptr operand+4, 1

                rcr     word ptr operand+4, 1
                rcr     word ptr operand+2, 1
                rcr     word ptr operand, 1
```

The only difference between this code and the code for the extended precision shift operations is that the first instruction is a rcl or rcr rather than a shl or shr instruction.

Performing an extended precision rol or ror instruction isn't quite as simple an operation. The 8086 extended precision versions of these instructions appear in the exercises. On the 80386 and later processors, you can use the bt, shld, and shrd instructions to easily implement an extended precision rol or ror instruction. The following code shows how to use the shld instruction to do an extended precision rol:

```
; Compute ROL EDX:EAX, 4

                mov     ebx, edx
                shld    edx, eax, 4
                shld    eax, ebx, 4
                bt      eax, 0      ;Set carry flag, if desired.
```

An extended precision ror instruction is similar; just keep in mind that you work on the L.O. end of the object first and the H.O. end last.

9.4 Operating on Different Sized Operands

Occasionally you may need to compute some value on a pair of operands that are not the same size. For example, you may need to add a word and a double word together or subtract a byte value from a word value. The solution is simple: just extend the smaller operand to the size of the larger operand and then do the operation on two similarly sized operands. For signed operands, you would sign extend the smaller operand to the same size as the larger operand; for unsigned values, you zero extend the smaller operand. This works for any operation, although the following examples demonstrate this for the addition operation.

To extend the smaller operand to the size of the larger operand, use a sign extension or zero extension operation (depending upon whether you're adding signed or unsigned values). Once you've extended the smaller value to the size of the larger, the addition can proceed. Consider the following code that adds a byte value to a word value:

```
var1            byte    ?
var2            word    ?

Unsigned addition:          Signed addition:

mov    al, var1             mov       al, var1
mov    ah, 0                cbw
add    ax, var2             add       ax, var2
```

In both cases, the byte variable was loaded into the al register, extended to 16 bits, and then added to the word operand. This code works out really well if you can choose the order of the operations (e.g., adding the eight bit value to the sixteen bit value). Sometimes, you cannot specify the order of the operations. Perhaps the sixteen bit value is already in the ax register and you want to add an eight bit value to it. For unsigned addition, you could use the following code:

```
                mov     ax, var2    ;Load 16 bit value into AX
                 .                  ;Do some other operations leaving
                 .                  ; a 16 bit quantity in AX.
                add     al, var1    ;Add in the 8 bit value.
                adc     ah, 0       ;Add carry into the H.O. word.
```

The first add instruction in this example adds the byte at var1 to the L.O. byte of the value in the accumulator. The adc instruction above adds the carry out of the L.O. byte into the H.O. byte of the accumulator. Care must be taken to ensure that this adc instruction is present. If you leave it out, you may not get the correct result.

Adding an eight bit signed operand to a sixteen bit signed value is a little more difficult. Unfortunately, you cannot add an immediate value (as above) to the H.O. word of ax. This is because the H.O. extension byte can be either 00h or 0FFh. If a register is available, the best thing to do is the following:

```
                mov     bx, ax      ;BX is the available register.
                mov     al, var1
                cbw
                add     ax, bx
```

If an extra register is not available, you might try the following code:

```
                add     al, var1
                cmp     var1, 0
                jge     add0
                adc     ah, 0FFh
                jmp     addedFF
add0:           adc     ah, 0
addedFF:
```

Of course, if another register isn't available, you could always push one onto the stack and save it while you're performing the operation, e.g.,

```
push    bx
mov     bx, ax
mov     al, var1
cbw
add     ax, bx
pop     bx
```

Another alternative is to store the 16 bit value in the accumulator into a memory location and then proceed as before:

```
mov     temp, ax
mov     al, var1
cbw
add     ax, temp
```

All the examples above added a byte value to a word value. By zero or sign extending the smaller operand to the size of the larger operand, you can easily add any two different sized variables together. Consider the following code that adds a signed byte operand to a signed double word:

```
var1            byte    ?
var2            dword   ?

                mov     al, var1
                cbw
                cwd                             ;Extend to 32 bits in DX
                add     ax, word ptr var2
                adc     dx, word ptr var2+2
```

Of course, if you have an 80386 or later processor, you could use the following code:

```
                movsx   eax, var1
                add     eax, var2
```

An example more applicable to the 80386 is adding an eight bit value to a quadword (64 bit) value, consider the following code:

```
BVal            byte    -1
QVal            qword     1

                movsx   eax, BVal
                cdq
                add     eax, dword ptr QVal
                adc     edx, dword ptr QVal+4
```

For additional examples, see the exercises at the end of this chapter.

9.5 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncracies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as "tricky programming" that you should always avoid in well written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well-known and commonly found in assembly language programs. Some of them can be really tricky, but a good number of them are simply "tricks of the trade." This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

9.5.1 Multiplying Without MUL and IMUL

If you take a quick look at the timing for the multiply instruction, you'll notice that the execution time for this instruction is rather long. Only the div and idiv instructions take longer on the 8086. When multiplying by a constant, you can avoid the performance penalty of the mul and imul instructions by using shifts, additions, and subtractions to perform the multiplication.

Remember, a shl operation performs the same operation as multiplying the specified operand by two. Shifting to the left two bit positions multiplies the operand by four. Shifting to the left three bit positions multiplies the operand by eight. In general, shifting an operand to the left n bits multiplies it by 2^n. Any value can be multiplied by some constant using a series of shifts and adds or shifts and subtractions. For example, to multiply the ax register by ten, you need only multiply it by eight and then add in two times the original value. That is, 10*ax = 8*ax + 2*ax. The code to accomplish this is

```
shl     ax, 1       ;Multiply AX by two
mov     bx, ax      ;Save 2*AX for later
shl     ax, 1       ;Multiply AX by four
shl     ax, 1       ;Multiply AX by eight
add     ax, bx      ;Add in 2*AX to get 10*AX
```

The ax register (or just about any register, for that matter) can be multiplied by most constant values much faster using shl than by using the mul instruction. This may seem hard to believe since it only takes two instructions to compute this product:

```
mov     bx, 10
mul     bx
```

However, if you look at the timings, the shift and add example above requires fewer clock cycles on most processors in the 80x86 family than the mul instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it. Of course, on the later 80x86 processors, the mul instruction is quite a bit faster than the earlier processors, but the shift and add scheme is generally faster on these processors as well.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by seven:

```
mov     bx, ax      ;Save AX*1
shl     ax, 1       ;AX := AX*2
shl     ax, 1       ;AX := AX*4
shl     ax, 1       ;AX := AX*8
sub     ax, bx      ;AX := AX*7
```

This follows directly from the fact that ax*7 = (ax*8)-ax.

A common error made by beginning assembly language students is subtracting or adding one or two rather than ax*1 or ax*2. The following does *not* compute ax*7:

```
shl     ax, 1
shl     ax, 1
shl     ax, 1
sub     ax, 1
```

It computes (8*ax)-1, something entirely different (unless, of course, ax = 1). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

You can also use the lea instruction to compute certain products on 80386 and later processors. The trick is to use the 80386 scaled index mode. The following examples demonstrate some simple cases:

```
lea     eax, [ecx][ecx]         ;EAX := ECX * 2
lea     eax, [eax]eax*2]        ;EAX := EAX * 3
lea     eax, [eax*4]            ;EAX := EAX * 4
lea     eax, [ebx][ebx*4]       ;EAX := EBX * 5
lea     eax, [eax*8]            ;EAX := EAX * 8
```

```
lea     eax, [edx][edx*8]       ;EAX := EDX * 9
```

9.5.2 Division Without DIV and IDIV

Much as the shl instruction can be used for simulating a multiplication by some power of two, the shr and sar instructions can be used to simulate a division by a power of two. Unfortunately, you cannot use shifts, additions, and subtractions to perform a division by an arbitrary constant as easily as you can use these instructions to perform a multiplication operation.

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. The multiply instruction is marginally faster than the divide instruction; multiplying by a reciprocal is almost always faster than division.

Now you're probably wondering "how does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by one tenth, there is no way you can load the value $1/10^{th}$ into an 80x86 register prior to performing the division. However, we could multiply $1/10^{th}$ by 10, perform the multiplication, and then divide the result by ten to get the final result. Of course, this wouldn't buy you anything at all, in fact it would make things worse since you're now doing a multiplication by ten as well as a division by ten. However, suppose you multiply 1/10th by 65,536 (6553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides ax by ten:

```
mov     dx, 6554    ;Round (65,536/10)
mul     dx
```

This code leaves ax/10 in the dx register.

To understand how this works, consider what happens when you multiply ax by 65,536 (10000h). This simply moves ax into dx and sets ax to zero. Multiplying by 6,554 (65,536 divided by ten) puts ax divided by ten into the dx register. Since mul is marginally faster than div, this technique runs a little faster than using a straight division.

Multiplying by the reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal only pays off if you perform the division many, many times (by the same value).

9.5.3 Using AND to Compute Remainders

The and instruction can be used to quickly compute remainders of the form:

```
dest := dest MOD 2^n
```

To compute a remainder using the and instruction, simply and the operand with the value 2^n-1. For example, to compute ax = ax mod 8 simply use the instruction:

```
and     ax, 7
```

Additional examples:

```
and     ax, 3       ;AX := AX mod 4
and     ax, 0Fh     ;AX := AX mod 16
and     ax, 1Fh     ;AX := AX mod 32
and     ax, 3Fh     ;AX := AX mod 64
and     ax, 7Fh     ;AX := AX mod 128
mov     ah, 0       ;AX := AX mod 256
                    ; (Same as ax and 0FFh)
```

9.5.4 Implementing Modulo-n Counters with AND

If you want to implement a counter variable that counts up to 2^n-1 and then resets to zero, simply using the following code:

```
inc     CounterVar
and     CounterVar, nBits
```

where nBits is a binary value containing n one bits right justified in the number. For example, to create a counter that cycles between zero and fifteen, you could use the following:

```
inc     CounterVar
and     CounterVar, 00001111b
```

9.5.5 Testing an Extended Precision Value for 0FFFF..FFh

The and instruction can be used to quickly check a multi-word value to see if it contains ones in all its bit positions. Simply load the first word into the ax register and then logically and the ax register with all the remaining words in the data structure. When the and operation is complete, the ax register will contain 0FFFFh if and only if all the words in that structure contained 0FFFFh. E.g.,

```
mov     ax, word ptr var
and     ax, word ptr var+2
and     ax, word ptr var+4
  .
  .
  .
and     ax, word ptr var+n
cmp     ax, 0FFFFh
je      Is0FFFFh
```

9.5.6 TEST Operations

Remember, the test instruction is an and instruction that doesn't retain the results of the and operation (other than the flag settings). Therefore, many of the comments concerning the and operation (particularly with respect to the way it affects the flags) also hold for the test instruction. However, since the test instruction doesn't affect the destination operand, multiple bit tests may be performed on the same value. Consider the following code:

```
test    ax, 1
jnz     Bit0
test    ax, 2
jnz     Bit1
test    ax, 4
jnz     Bit3
etc.
```

This code can be used to successively test each bit in the ax register (or any other operand for that matter). Note that you cannot use the test/cmp instruction pair to test for a specific value within a string of bits (as you can with the and/cmp instructions). Since test doesn't strip out any unwanted bits, the cmp instruction would actually be comparing the original value rather than the stripped value. For this reason, you'll normally use the test instruction to see if a single bit is set or if one or more bits out of a group of bits are set. Of course, if you have an 80386 or later processor, you can also use the bt instruction to test individual bits in an operand.

Another important use of the test instruction is to efficiently compare a register against zero. The following test instruction sets the zero flag if and only if ax contains zero (anything anded with itself produces its original value; this sets the zero flag only if that value is zero):

```
test ax, ax
```

The test instruction is shorter than

```
                cmp     ax, 0
```

or

```
                cmp     eax, 0
```

though it is no better than

```
                cmp     al, 0
```

Note that you can use the and and or instructions to test for zero in a fashion identical to test. However, on pipelined processors like the 80486 and Pentium chips, the test instruction is less likely to create a hazard since it does not store a result back into its destination register.

9.5.7 Testing Signs with the XOR Instruction

Remember the pain associated with a multi-precision signed multiplication operation? You need to determine the sign of the result, take the absolute value of the operands, multiply them, and then adjust the sign of the result as determined before the multiplication operation. The sign of the product of two numbers is simply the exclusive-or of their signs before performing the multiplication. Therefore, you can use the xor instruction to determine the sign of the product of two extended precision numbers. E.g.,

```
32x32 Multiply:
                mov     al, byte ptr Oprnd1+3
                xor     al, byte ptr Oprnd2+3
                mov     cl, al                  ;Save sign

; Do the multiplication here (don't forget to take the absolute
; value of the two operands before performing the multiply).

                .
                .
                .

; Now fix the sign.

                cmp     cl, 0                   ;Check sign bit
                jns     ResultIsPos

; Negate the product here.

                .
                .
                .

ResultIsPos:
```

9.6 Masking Operations

A *mask* is a value used to force certain bits to zero or one within some other value. A mask typically affects certain bits in an operand (forcing them to zero or one) and leaves other bits unaffected. The appropriate use of masks allows you to *extract* bits from a value, *insert* bits into a value, and pack or unpacked a packed data type. The following sections describe these operations in detail.

9.6.1 Masking Operations with the AND Instruction

If you'll take a look at the truth table for the and operation back in Chapter One, you'll note that if you fix either operand at zero the result is always zero. If you set that operand to one, the result is always the value of the other operand. We can use this property of the and instruction to selectively force certain bits to zero in a value without affecting other bits. This is called *masking out* bits.

As an example, consider the ASCII codes for the digits “0”..”9”. Their codes fall in the range 30h..39h respectively. To convert an ASCII digit to its corresponding numeric value, you must subtract 30h from the ASCII code. This is easily accomplished by logically anding the value with 0Fh. This strips (sets to zero) all but the L.O. four bits producing the numeric value. You could have used the subtract instruction, but most people use the and instruction for this purpose.

9.6.2 Masking Operations with the OR Instruction

Much as you can use the and instruction to force selected bits to zero, you can use the or instruction to force selected bits to one. This operation is called *masking in bits*.

Remember the masking out operation described earlier with the and instruction? In that example we wanted to convert an ASCII code for a digit to its numeric equivalent. You can use the or instruction to reverse this process. That is, convert a numeric value in the range 0..9 to the ASCII code for the corresponding digit, i.e., ‘0’..’9’. To do this, logically or the specified numeric value with 30h.

9.7 Packing and Unpacking Data Types

One of the primary uses of the shift and rotate instructions is packing and unpacking data. Byte and word data types are chosen more often than any other since the 80x86 supports these two data sizes with hardware. If you don’t need exactly eight or 16 bits, using a byte or word to hold your data might be wasteful. By packing data, you may be able to reduce memory requirements for your data by inserting two or more values into a single byte or word. The cost for this reduction in memory use is lower performance. It takes time to pack and unpack the data. Nevertheless, for applications that aren’t speed critical (or for those portions of the application that aren’t speed critical), the memory savings might justify the use of packed data.

The data type that offers the most savings when using packing techniques is the boolean data type. To represent true or false requires a single bit. Therefore, up to eight different boolean values can be packed into a single byte. This represents an 8:1 compression ratio, therefore, a packed array of boolean values requires only one-eighth the space of an equivalent unpacked array (where each boolean variable consumes one byte). For example, the Pascal array

```
B:packed array[0..31] of boolean;
```

requires only four bytes when packed one value per bit. When packed one value per byte, this array requires 32 bytes.

Dealing with a packed boolean array requires two operations. You’ll need to insert a value into a packed variable (often called a packed field) and you’ll need to extract a value from a packed field.

To insert a value into a packed boolean array, you must align the source bit with its position in the destination operand and then store that bit into the destination operand. You can do this with a sequence of and, or, and shift instructions. The first step is to mask out the corresponding bit in the destination operand. Use an and instruction for this. Then the source operand is shifted so that it is aligned with the destination position, finally the source operand is or’d into the destination operand. For example, if you want to insert bit zero of the ax register into bit five of the cx register, the following code could be used:

```
and     cl, 0DFh    ;Clear bit five (the destination bit)
and     al, 1       ;Clear all AL bits except the src bit.
ror     al, 1       ;Move to bit 7
shr     al, 1       ;Move to bit 6
shr     al, 1       ;move to bit 5
or      cl, al
```

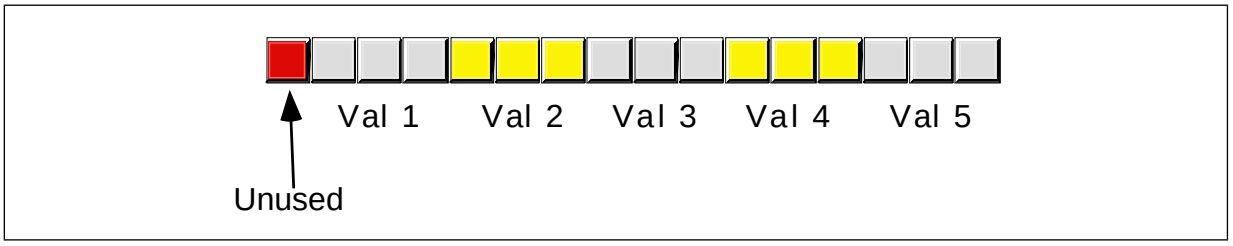


Figure 8.4 Packed Data

This code is somewhat tricky. It rotates the data to the right rather than shifting it to the left since this requires fewer shifts and rotate instructions.

To extract a boolean value, you simply reverse this process. First, you move the desired bit into bit zero and then mask out all the other bits. For example, to extract the data in bit five of the cx register leaving the single boolean value in bit zero of the ax register, you'd use the following code:

```
mov     al, cl
shl     al, 1       ;Bit 5 to bit 6
shl     al, 1       ;Bit 6 to bit 7
rol     al, 1       ;Bit 7 to bit 0
and     ax, 1       ;Clear all bits except 0
```

To test a boolean variable in a packed array you needn't extract the bit and then test it, you can test it in place. For example, to test the value in bit five to see if it is zero or one, the following code could be used:

```
test    cl, 00100000b
jnz     BitIsSet
```

Other types of packed data can be handled in a similar fashion except you need to work with two or more bits. For example, suppose you've packed five different three bit fields into a sixteen bit value as shown in Figure 8.4.

If the ax register contains the data to pack into value3, you could use the following code to insert this data into field three:

```
mov     ah, al       ;Do a shl by 8
shr     ax, 1        ;Reposition down to bits 6..8
shr     ax, 1
and     ax, 11100000b ;Strip undesired bits
and     DATA, 0FE3Fh  ;Set destination field to zero.
or      DATA, ax      ;Merge new data into field.
```

Extraction is handled in a similar fashion. First you strip the unneeded bits and then you justify the result:

```
mov     ax, DATA
and     ax, 1Ch
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
shr     ax, 1
```

This code can be improved by using the following code sequence:

```
mov     ax, DATA
shl     ax, 1
shl     ax, 1
mov     al, ah
and     ax, 07h
```

Additional uses for packed data will be explored throughout this book.

9.8 Tables

The term “table” has different meanings to different programmers. To most assembly language programmers, a table is nothing more than an array that is initialized with some data. The assembly language programmer often uses tables to compute complex or otherwise slow functions. Many very high level languages (e.g., SNOBOL4 and Icon) directly support a table data type. Tables in these languages are essentially arrays whose elements you can access with an non-integer value (e.g., floating point, string, or any other data type). In this section, we will adopt the assembly language programmer’s view of tables.

A Table is an array containing preinitialized values that do not change during the execution of the program. A table can be compared to an array in the same way an integer constant can be compared to an integer variable. In assembly language, you can use tables for a variety of purposes: computing functions, controlling program flow, or simply “looking things up”. In general, tables provide a fast mechanism for performing some operation at the expense of some space in your program (the extra space holds the tabular data). In the following sections we’ll explore some of the many possible uses of tables in an assembly language program.

9.8.1 Function Computation via Table Look Up

Tables can do all kinds of things in assembly language. In HLLs, like Pascal, it’s real easy to create a formula which computes some value. A simple looking arithmetic expression is equivalent to a considerable amount of 80x86 assembly language code. Assembly language programmers tend to compute many values via table look up rather than through the execution of some function. This has the advantage of being easier, and often more efficient as well. Consider the following Pascal statement:

if (character >= ‘a’) and (character <= ‘z’) then character := chr(ord(character) - 32);

This Pascal if statement converts the character variable character from lower case to upper case if character is in the range ‘a’..’z’. The 80x86 assembly language code that does the same thing is

```
                mov     al, character
                cmp     al, 'a'
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 05fh     ;Same operation as SUB AL,32
NotLower:       mov     character, al
```

Had you buried this code in a nested loop, you’d be hard pressed to improve the speed of this code without using a table look up. Using a table look up, however, allows you to reduce this sequence of instructions to just four instructions:

```
                mov     al, character
                lea     bx, CnvrtLower
                xlat
                mov     character, al
```

CnvrtLower is a 256-byte table which contains the values 0..60h at indices 0..60h, 41h..5Ah at indices 61h..7Ah, and 7Bh..0FFh at indices 7Bh..0FFh. Often, using this table look up facility will increase the speed of your code.

As the complexity of the function increases, the performance benefits of the table look up method increase dramatically. While you would almost never use a look up table to convert lower case to upper case, consider what happens if you want to swap cases:

Via computation:

```
                mov     al, character
                cmp     al, 'a'
```

```
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 05fh
                jmp     ConvertDone

NotLower:       cmp     al, 'A'
                jb      ConvertDone
                cmp     al, 'Z'
                ja      ConvertDone
                or      al, 20h
ConvertDone:
                mov     character, al
```

The table look up code to compute this same function is:

```
                mov     al, character
                lea     bx, SwapUL
                xlat
                mov     character, al
```

As you can see, when computing a function via table look up, no matter what the function is, only the table changes, not the code doing the look up.

Table look ups suffer from one major problem – functions computed via table look ups have a limited domain. The domain of a function is the set of possible input values (parameters) it will accept. For example, the upper/lower case conversion functions above have the 256-character ASCII character set as their domain.

A function such as SIN or COS accepts the set of real numbers as possible input values. Clearly the domain for SIN and COS is much larger than for the upper/lower case conversion function. If you are going to do computations via table look up, you must limit the domain of a function to a small set. This is because each element in the domain of a function requires an entry in the look up table. You won't find it very practical to implement a function via table look up whose domain the set of real numbers.

Most look up tables are quite small, usually 10 to 128 entries. Rarely do look up tables grow beyond 1,000 entries. Most programmers don't have the patience to create (and verify the correctness) of a 1,000 entry table.

Another limitation of functions based on look up tables is that the elements in the domain of the function must be fairly contiguous. Table look ups take the input value for a function, use this input value as an index into the table, and return the value at that entry in the table. If you do not pass a function any values other than 0, 100, 1,000, and 10,000 it would seem an ideal candidate for implementation via table look up, its domain consists of only four items. However, the table would actually require 10,001 different elements due to the range of the input values. Therefore, you cannot efficiently create such a function via a table look up. Throughout this section on tables, we'll assume that the domain of the function is a fairly contiguous set of values.

The best functions that can be implemented via table look ups are those whose domain and range is always 0..255 (or some subset of this range). Such functions are efficiently implemented on the 80x86 via the XLAT instruction. The upper/lower case conversion routines presented earlier are good examples of such a function. Any function in this class (those whose domain and range take on the values 0..255) can be computed using the same two instructions (lea bx,table / xlat) above. The only thing that ever changes is the look up table.

The xlat instruction cannot be (conveniently) used to compute a function value once the range or domain of the function takes on values outside 0..255. There are three situations to consider:

- The domain is outside 0..255 but the range is within 0..255,
- The domain is inside 0..255 but the range is outside 0..255, and
- Both the domain and range of the function take on values outside 0..255.

We will consider each of these cases separately.

If the domain of a function is outside 0..255 but the range of the function falls within this set of values, our look up table will require more than 256 entries but we can represent each entry with a single byte. Therefore, the look up table can be an array of bytes. Next to look ups involving the xlat instruction, functions falling into this class are the most efficient. The following Pascal function invocation,

```
B := Func(X);
```

where Func is

```
function Func(X:word):byte;
```

consists of the following 80x86 code:

```
mov     bx, X
mov     al, FuncTable [bx]
mov     B, al
```

This code loads the function parameter into bx, uses this value (in the range 0..??) as an index into the FuncTable table, fetches the byte at that location, and stores the result into B. Obviously, the table must contain a valid entry for each possible value of X. For example, suppose you wanted to map a cursor position on the video screen in the range 0..1999 (there are 2,000 character positions on an 80x25 video display) to its X or Y coordinate on the screen. You could easily compute the X coordinate via the function X:=Posn mod 80 and the Y coordinate with the formula Y:=Posn div 80 (where Posn is the cursor position on the screen). This can be easily computed using the 80x86 code:

```
mov     bl, 80
mov     ax, Posn
div     bx

; X is now in AH, Y is now in AL
```

However, the div instruction on the 80x86 is very slow. If you need to do this computation for every character you write to the screen, you will seriously degrade the speed of your video display code. The following code, which realizes these two functions via table look up, would improve the performance of your code considerably:

```
mov     bx, Posn
mov     al, YCoord[bx]
mov     ah, XCoord[bx]
```

If the domain of a function is within 0..255 but the range is outside this set, the look up table will contain 256 or fewer entries but each entry will require two or more bytes. If both the range and domains of the function are outside 0..255, each entry will require two or more bytes and the table will contain more than 256 entries.

Recall from Chapter Four the formula for indexing into a single dimensional array (of which a table is a special case):

```
Address := Base + index * size
```

If elements in the range of the function require two bytes, then the index must be multiplied by two before indexing into the table. Likewise, if each entry requires three, four, or more bytes, the index must be multiplied by the size of each table entry before being used as an index into the table. For example, suppose you have a function, F(x), defined by the following (pseudo) Pascal declaration:

```
function F(x:0..999):word;
```

You can easily create this function using the following 80x86 code (and, of course, the appropriate table):

```
mov     bx, X       ;Get function input value and
shl     bx, 1       ; convert to a word index into F.
mov     ax, F[bx]
```

The shl instruction multiplies the index by two, providing the proper index into a table whose elements are words.

Any function whose domain is small and mostly contiguous is a good candidate for computation via table look up. In some cases, non-contiguous domains are acceptable as well, as long as the domain can be coerced into an appropriate set of values. Such operations are called conditioning and are the subject of the next section.

9.8.2 Domain Conditioning

Domain conditioning is taking a set of values in the domain of a function and massaging them so that they are more acceptable as inputs to that function. Consider the following function:

$$\sin x = \langle \sin x | x \in [-2\pi, 2\pi] \rangle$$

This says that the (computer) function SIN(x) is equivalent to the (mathematical) function *sin x* where

$$-2\pi \le x \le 2\pi$$

As we all know, sine is a circular function which will accept any real valued input. The formula used to compute sine, however, only accept a small set of these values.

This range limitation doesn't present any real problems, by simply computing SIN(X mod (2*pi)) we can compute the sine of any input value. Modifying an input value so that we can easily compute a function is called conditioning the input. In the example above we computed X mod 2*pi and used the result as the input to the sin function. This truncates X to the domain sin needs without affecting the result. We can apply input conditioning can be applied to table look ups as well. In fact, scaling the index to handle word entries is a form of input conditioning. Consider the following Pascal function:

```
function val(x:word):word; begin
        case x of
                0: val := 1;
                1: val := 1;
                2: val := 4;
                3: val := 27;
                4: val := 256;
                otherwise val := 0;
        end;
end;
```

This function computes some value for x in the range 0..4 and it returns zero if x is outside this range. Since x can take on 65,536 different values (being a 16 bit word), creating a table containing 65,536 words where only the first five entries are non-zero seems to be quite wasteful. However, we can still compute this function using a table look up if we use input conditioning. The following assembly language code presents this principle:

```
                xor     ax, ax      ;AX := 0, assume X > 4.
                mov     bx, x
                cmp     bx, 4
                ja      ItsZero
                shl     bx, 1
                mov     ax, val[bx]
ItsZero:
```

This code checks to see if x is outside the range 0..4. If so, it manually sets ax to zero, otherwise it looks up the function value through the val table. With input conditioning, you can implement several functions that would otherwise be impractical to do via table look up.

9.8.3 Generating Tables

One big problem with using table look ups is creating the table in the first place. This is particularly true if there are a large number of entries in the table. Figuring out the data to place in the table, then laboriously entering the data, and, finally, checking that data to make sure it is valid, is a very time-staking and boring process. For many tables, there is no way around this process. For other tables there is a better way – use the computer to generate the table for you. An example is probably the best way to describe this. Consider the following modification to the sine function:

$$(\sin x) \times r = \langle \frac{(r \times (1000 \times \sin x))}{1000} | x \in [0,$$

This states that x is an integer in the range 0..359 and r is an integer. The computer can easily compute this with the following code:

```
mov     bx, X
shl     bx, 1
mov     ax, Sines [bx] ;Get SIN(X)*1000
mov     bx, R          ;Compute R*(SIN(X)*1000)
mul     bx
mov     bx, 1000       ;Compute (R*(SIN(X)*1000))/1000
div     bx
```

Note that integer multiplication and division are not associative. You cannot remove the multiplication by 1000 and the division by 1000 because they seem to cancel one another out. Furthermore, this code must compute this function in exactly this order. All that we need to complete this function is a table containing 360 different values corresponding to the sine of the angle (in degrees) times 1,000. Entering a table into an assembly language program containing such values is extremely boring and you'd probably make several mistakes entering and verifying this data. However, you can have the program generate this table for you. Consider the following Turbo Pascal program:

```
program maketable;
var    i:integer;
       r:integer;
       f:text;
begin
       assign(f,'sines.asm');
       rewrite(f);
       for i := 0 to 359 do begin
                 r := round(sin(I * 2.0 * pi / 360.0) * 1000.0);
                 if (i mod 8) = 0 then begin
                          writeln(f);
                          write(f,' dw ',r);
                 end
                 else write(f,',',r);
       end;
       close(f);
 end.
```

This program produces the following output:

```
dw 0,17,35,52,70,87,105,122
dw 139,156,174,191,208,225,242,259
dw 276,292,309,326,342,358,375,391
dw 407,423,438,454,469,485,500,515
dw 530,545,559,574,588,602,616,629
dw 643,656,669,682,695,707,719,731
dw 743,755,766,777,788,799,809,819
dw 829,839,848,857,866,875,883,891
dw 899,906,914,921,927,934,940,946
dw 951,956,961,966,970,974,978,982
dw 985,988,990,993,995,996,998,999
dw 999,1000,1000,1000,999,999,998,996
dw 995,993,990,988,985,982,978,974
dw 970,966,961,956,951,946,940,934
dw 927,921,914,906,899,891,883,875
```

```
dw 866,857,848,839,829,819,809,799
dw 788,777,766,755,743,731,719,707
dw 695,682,669,656,643,629,616,602
dw 588,574,559,545,530,515,500,485
dw 469,454,438,423,407,391,375,358
dw 342,326,309,292,276,259,242,225
dw 208,191,174,156,139,122,105,87
dw 70,52,35,17,0,-17,-35,-52
dw -70,-87,-105,-122,-139,-156,-174,-191
dw -208,-225,-242,-259,-276,-292,-309,-326
dw -342,-358,-375,-391,-407,-423,-438,-454
dw -469,-485,-500,-515,-530,-545,-559,-574
dw -588,-602,-616,-629,-643,-656,-669,-682
dw -695,-707,-719,-731,-743,-755,-766,-777
dw -788,-799,-809,-819,-829,-839,-848,-857
dw -866,-875,-883,-891,-899,-906,-914,-921
dw -927,-934,-940,-946,-951,-956,-961,-966
dw -970,-974,-978,-982,-985,-988,-990,-993
dw -995,-996,-998,-999,-999,-1000,-1000,-1000
dw -999,-999,-998,-996,-995,-993,-990,-988
dw -985,-982,-978,-974,-970,-966,-961,-956
dw -951,-946,-940,-934,-927,-921,-914,-906
dw -899,-891,-883,-875,-866,-857,-848,-839
dw -829,-819,-809,-799,-788,-777,-766,-755
dw -743,-731,-719,-707,-695,-682,-669,-656
dw -643,-629,-616,-602,-588,-574,-559,-545
dw -530,-515,-500,-485,-469,-454,-438,-423
dw -407,-391,-375,-358,-342,-326,-309,-292
dw -276,-259,-242,-225,-208,-191,-174,-156
dw -139,-122,-105,-87,-70,-52,-35,-17
```

Obviously it's much easier to write the Turbo Pascal program that generated this data than to enter (and verify) this data by hand. This little example shows how useful Pascal can be to the assembly language programmer!

9.9 Sample Programs

This chapter's sample programs demonstrate several important concepts including extended precision arithmetic and logical operations, arithmetic expression evaluation, boolean expression evaluation, and packing/unpacking data.

9.9.1 Converting Arithmetic Expressions to Assembly Language

The following sample program (Pgm9_1.asm on the companion CD-ROM) provides some examples of converting arithmetic expressions into assembly language:

```
; Pgm9_1.ASM
;
; Several examples demonstrating how to convert various
; arithmetic expressions into assembly language.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment  para public 'data'

; Arbitrary variables this program uses.

u               word     ?
v               word     ?
w               word     ?
x               word     ?
y               word     ?
```

```
dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; GETI-Reads an integer variable from the user and returns its
;      its value in the AX register.

geti            textequ  <call _geti>
_geti           proc
                push     es
                push     di

                getsm
                atoi
                free

                pop      di
                pop      es
                ret
_geti           endp


Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                print
                byte     "Abitrary expression program",cr,lf
                byte     "---------------------------",cr,lf
                byte     lf
                byte     "Enter a value for u: ",0

                geti
                mov      u, ax

                print
                byte     "Enter a value for v: ",0
                geti
                mov      v, ax

                print
                byte     "Enter a value for w: ",0
                geti
                mov      w, ax

                print
                byte     "Enter a non-zero value for x: ",0
                geti
                mov      x, ax

                print
                byte     "Enter a non-zero value for y: ",0
                geti
                mov      y, ax

; Okay, compute Z := (X+Y)*(U+V*W)/X and print the result.

                print
                byte     cr,lf
                byte     "(X+Y) * (U+V*W)/X is ",0

                mov      ax, v        ;Compute V*W
                imul     w            ; and then add in
                add      ax, u        ; U.
                mov      bx, ax       ;Save in a temp location for now.

                mov      ax, x        ;Compute X+Y, multiply this
                add      ax, y        ; sum by the result above,
                imul     bx           ; and then divide the whole
```

```
                idiv    x           ; thing by X.

                puti
                putcr

; Compute ((X-Y*U) + (U*V) - W)/(X*Y)

                print
                byte    "((X-Y*U) + (U*V) - W)/(X*Y) = ",0

                mov     ax, y       ;Compute y*u first
                imul    u
                mov     dx, X       ;Now compute X-Y*U
                sub     dx, ax
                mov     cx, dx      ;Save in temp

                mov     ax, u       ;Compute U*V
                imul    V
                add     cx, ax      ;Compute (X-Y*U) + (U*V)

                sub     cx, w       ;Compute ((X-Y*U) + (U*V) - W)

                mov     ax, x       ;Compute (X*Y)
                imul    y

                xchg    ax, cx
                cwd                 ;Compute NUMERATOR/(X*Y)
                idiv    cx

                puti
                putcr

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

9.9.2 Boolean Operations Example

The following sample program (Pgm9_2.asm on the companion CD-ROM) demonstrates how to manipulate boolean values in assembly language. It also provides an example of Demorgan's Theorems in operation.

```
; Pgm9_2.ASM
;
; This program demonstrates DeMorgan's theorems and
; various other logical computations.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment  para public 'data'


; Boolean input variables for the various functions
; we are going to test.
```

```
a               byte    0
b               byte    0

dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

; Get0or1-Reads a "0" or "1" from the user and returns its
;               its value in the AX register.

get0or1         textequ <call _get0or1>
_get0or1        proc
                push    es
                push    di

                getsm
                atoi
                free

                pop     di
                pop     es
                ret
_get0or1        endp


Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                print
                byte    "Demorgan's Theorems",cr,lf
                byte    "-------------------",cr,lf
                byte    lf
                byte    "According to Demorgan's theorems, all results "
                byte    "between the dashed lines",cr,lf
                byte    "should be equal.",cr,lf
                byte    lf
                byte    "Enter a value for a: ",0

                get0or1
                mov     a, al

                print
                byte    "Enter a value for b: ",0
                get0or1
                mov     b, al

                print
                byte    "---------------------------------",cr,lf
                byte    "Computing not (A and B): ",0

                mov     ah, 0
                mov     al, a
                and     al, b
                xor     al, 1       ;Logical NOT operation.

                puti
                putcr

                print
                byte    "Computing (not A) OR (not B): ",0
                mov     al, a
                xor     al, 1
                mov     bl, b
                xor     bl, 1
                or      al, bl
```

```
                        puti

                        print
                        byte    cr,lf
                        byte    "--------------------------------",cr,lf
                        byte    "Computing (not A) OR B: ",0
                        mov     al, a
                        xor     al, 1
                        or      al, b
                        puti

                        print
                        byte    cr,lf
                        byte    "Computing not (A AND (not B)): ",0
                        mov     al, b
                        xor     al, 1
                        and     al, a
                        xor     al, 1
                        puti

                        print
                        byte    cr,lf
                        byte    "--------------------------------",cr,lf
                        byte    "Computing (not A) OR B: ",0
                        mov     al, a
                        xor     al, 1
                        or      al, b
                        puti

                        print
                        byte    cr,lf
                        byte    "Computing not (A AND (not B)): ",0
                        mov     al, b
                        xor     al, 1
                        and     al, a
                        xor     al, 1
                        puti

                        print
                        byte    cr,lf
                        byte    "--------------------------------",cr,lf
                        byte    "Computing not (A OR B): ",0
                        mov     al, a
                        or      al, b
                        xor     al, 1
                        puti

                        print
                        byte    cr,lf
                        byte    "Computing (not A) AND (not B): ",0
                        mov     al, a
                        xor     al, 1
                        and     bl, b
                        xor     bl, 1
                        and     al, bl
                        puti

                        print
                        byte    cr,lf
                        byte    "--------------------------------",cr,lf
                        byte    0

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends
```

```
zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

9.9.3 64-bit Integer I/O

This sample program (Pgm9_3.asm on the companion CD-ROM) shows how to read and write 64-bit integers. It provides the ATOU64 and PUTU64 routines that let you convert a string of digits to a 64-bit unsigned integer and output a 64-bit unsigned integer as a decimal string to the display.

```
; Pgm9_3.ASM
;
; This sample program provides two procedures that read and write
; 64-bit unsigned integer values on an 80386 or later processor.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

                .386
                option   segment:use16

dp              textequ  <dword ptr>
byp             textequ  <byte ptr>

dseg            segment  para public 'data'

; Acc64 is a 64 bit value that the ATOU64 routine uses to input
; a 64-bit value.

Acc64           qword    0

; Quotient holds the result of dividing the current PUTU value by
; ten.

Quotient        qword    0

; NumOut holds the string of digits created by the PUTU64 routine.

NumOut          byte     32 dup (0)

; A sample test string for the ATOI64 routine:

LongNumber      byte     "123456789012345678",0

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; ATOU64-       On entry, ES:DI point at a string containing a
;               sequence of digits.  This routine converts that
;               string to a 64-bit integer and returns that
;               unsigned integer value in EDX:EAX.
;
;               This routine uses the algorithm:
;
;               Acc := 0
;               while digits left
;
;                       Acc := (Acc * 10) + (Current Digit - '0')
;                       Move on to next digit
;
```

```
;                endwhile

ATOU64           proc     near
                 push     di          ;Save because we modify it.
                 mov      dp Acc64, 0  ;Initialize our accumulator.
                 mov      dp Acc64+4, 0

; While we've got some decimal digits, process the input string:

                 sub      eax, eax     ;Zero out eax's H.O. 3 bytes.
WhileDigits:     mov      al, es:[di]
                 xor      al, '0'      ;Translates '0'..'9' -> 0..9
                 cmp      al, 10       ; and everything else is > 9.
                 ja       NotADigit

; Multiply Acc64 by ten.  Use shifts and adds to accomplish this:

                 shl      dp Acc64, 1            ;Compute Acc64*2
                 rcl      dp Acc64+4, 1

                 push     dp Acc64+4             ;Save Acc64*2
                 push     dp Acc64

                 shl      dp Acc64, 1            ;Compute Acc64*4
                 rcl      dp Acc64+4, 1
                 shl      dp Acc64, 1            ;Compute Acc64*8
                 rcl      dp Acc64+4, 1

                 pop      edx                    ;Compute Acc64*10 as
                 add      dp Acc64, edx          ; Acc64*2 + Acc64*8
                 pop      edx
                 adc      dp Acc64+4, edx

; Add in the numeric equivalent of the current digit.
; Remember, the H.O. three words of eax contain zero.

                 add      dp Acc64, eax          ;Add in this digit

                 inc      di                     ;Move on to next char.
                 jmp      WhileDigits            ;Repeat for all digits.

; Okay, return the 64-bit integer value in eax.

NotADigit:       mov      eax, dp Acc64
                 mov      edx, dp Acc64+4
                 pop      di
                 ret
ATOU64           endp

; PUTU64-        On entry, EDX:EAX contain a 64-bit unsigned value.
;                Output a string of decimal digits providing the
;                decimal representation of that value.
;
;                This code uses the following algorithm:
;
;                    di := 30;
;                    while edx:eax <> 0 do
;
;                       OutputNumber[di] := digit;
;                          edx:eax := edx:eax div 10
;                          di := di - 1;
;
;                    endwhile
;                    Output digits from OutNumber[di+1]
;                         through OutputNumber[30]

PUTU64           proc
                 push     es
                 push     eax
                 push     ecx
```

```
                push    edx
                push    di
                pushf



                mov     di, dseg                ;This is where the output
                mov     es, di                  ; string will go.
                lea     di, NumOut+30           ;Store characters in string
                std                             ; backwards.
                mov     byp es:[di+1],0         ;Output zero terminator.

; Save the value to print so we can divide it by ten using an
; extended precision division operation.

                mov     dp Quotient, eax
                mov     dp Quotient+4, edx

; Okay, begin converting the number into a string of digits.

                mov     ecx, 10                 ;Value to divide by.
DivideLoop:     mov     eax, dp Quotient+4      ;Do a 64-bit by
                sub     edx, edx                ; 32-bit division
                div     ecx                     ; (see the text
                mov     dp Quotient+4, eax      ;  for details).

                mov     eax, dp Quotient
                div     ecx
                mov     dp Quotient, eax

; At this time edx (dl, actually) contains the remainder of the
; above division by ten, so dl is in the range 0..9.  Convert
; this to an ASCII character and save it away.

                mov     al, dl
                or      al, '0'
                stosb

; Now check to see if the result is zero.  When it is, we can
; quit.

                mov     eax, dp Quotient
                or      eax, dp Quotient+4
                jnz     DivideLoop

OutputNumber:   inc     di
                puts
                popf
                pop     di
                pop     edx
                pop     ecx
                pop     eax
                pop     es
                ret
PUTU64          endp


; The main program provides a simple test of the two routines
; above.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                lesi    LongNumber
                call    ATOU64
                call    PutU64
                printf
```

```
                byte    cr,lf
                byte    "%x %x %x %x",cr,lf,0
                dword   Acc64+6, Acc64+4, Acc64+2, Acc64

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

9.9.4 Packing and Unpacking Date Data Types

This sample program demonstrates how to pack and unpack data using the Date data type introduced in Chapter One.

```
; Pgm9_4.ASM
;
;       This program demonstrates how to pack and unpack
;       data types.  It reads in a month, day, and year value.
;       It then packs these values into the format the textbook
;       presents in chapter two.  Finally, it unpacks this data
;       and calls the stdlib DTOA routine to print it as text.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment  para public 'data'

Month           byte     ? ;Holds month value (1-12)
Day             byte     ? ;Holds day value (1-31)
Year            byte     ? ;Holds year value (80-99)

Date            word     ? ;Packed data goes in here.

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; GETI-Reads an integer variable from the user and returns its
;       its value in the AX register.

geti            textequ  <call _geti>
_geti           proc
                push     es
                push     di

                getsm
                atoi
                free

                pop      di
                pop      es
                ret
_geti           endp
```

```
Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                print
                byte    "Date Conversion Program",cr,lf
                byte    "-----------------------",cr,lf
                byte    lf,0

; Get the month value from the user.
; Do a simple check to make sure this value is in the range
; 1-12.  Make the user reenter the month if it is not.

GetMonth:       print
                byte    "Enter the month (1-12): ",0

                geti
                mov     Month, al
                cmp     ax, 0
                je      BadMonth
                cmp     ax, 12
                jbe     GoodMonth
BadMonth:       print
                byte    "Illegal month value, please re-enter",cr,lf,0
                jmp     GetMonth

GoodMonth:

; Okay, read the day from the user.  Again, do a simple
; check to see if the date is valid.  Note that this code
; only checks to see if the day value is in the range 1-31.
; It does not check those months that have 28, 29, or 30
; day months.

GetDay:         print
                byte    "Enter the day (1-31): ",0
                geti
                mov     Day, al
                cmp     ax, 0
                je      BadDay
                cmp     ax, 31
                jbe     GoodDay
BadDay:         print
                byte    "Illegal day value, please re-enter",cr,lf,0
                jmp     GetDay

GoodDay:

; Okay, get the year from the user.
; This check is slightly more sophisticated.  If the user
; enters a year in the range 1980-1999, it will automatically
; convert it to 80-99.  All other dates outside the range
; 80-99 are illegal.

GetYear:        print
                byte    "Enter the year (80-99): ",0
                geti
                cmp     ax, 1980
                jb      TestYear
                cmp     ax, 1999
                ja      BadYear

                sub     dx, dx      ;Zero extend year to 32 bits.
                mov     bx, 100
                div     bx          ;Compute year mod 100.
                mov     ax, dx
                jmp     GoodYear
```

```
TestYear:       cmp     ax, 80
                jb      BadYear
                cmp     ax, 99
                jbe     GoodYear

BadYear:        print
                byte    "Illegal year value.  Please re-enter",cr,lf,0
                jmp     GetYear

GoodYear:       mov     Year, al

; Okay, take these input values and pack them into the following
; 16-bit format:
;
;      bit 15     8 7      0
;          |      | |      |
;         MMMMDDDD DYYYYYYY

                mov     ah, 0
                mov     bh, ah
                mov     al, Month   ;Put Month into bit positions
                mov     cl, 4       ; 12..15
                ror     ax, cl

                mov     bl, Day     ;Put Day into bit positions
                mov     cl, 7       ; 7..11.
                shl     bx, cl

                or      ax, bx      ;Create MMMMDDDD D0000000
                or      al, Year    ;Create MMMMDDDD DYYYYYYY
                mov     Date, ax    ;Save away packed date.

; Print out the packed date (in hex):

                print
                byte    "Packed date = ",0
                putw
                putcr

; Okay, the following code demonstrates how to unpack this date
; and put it in a form the standard library's LDTOAM routine can
; use.

                mov     ax, Date    ;First, extract Month
                mov     cl, 4
                shr     ah, cl
                mov     dh, ah      ;LDTOAM needs month in DH.

                mov     ax, Date    ;Next get the day.
                shl     ax, 1
                and     ah, 11111b
                mov     dl, ah      ;Day needs to be in DL.

                mov     cx, Date    ;Now process the year.
                and     cx, 7fh     ;Strip all but year bits.

                print
                byte    "Date: ",0
                LDTOAM              ;Convert to a string
                puts
                free
                putcr

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
```

```
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

9.10 Laboratory Exercises

In this laboratory you will perform the following activities:

- Use CodeView to set breakpoints within a program and locate some errors.
- Use CodeView to trace through sections of a program to discover problems with that program.
- Use CodeView to trace through some code you write to verify correctness and observe the calculation one step at a time.

9.10.1 Debugging Programs with CodeView

In past chapters of this lab manual you've had the opportunity to use CodeView to view the machine state (register and memory values), enter simple assembly language programs, and perform other minor tasks. In this section we will explore one of CodeView's most important capabilities - helping you locate problems within your code. This section discusses three features of CodeView we have ignored up to this point - Breakpoints, Watch operations, and code tracing. These features provide some very important tools for figuring out what is wrong with your assembly language programs.

Code tracing is a feature CodeView provides that lets you execute assembly language statements one at a time and observe the results. Many programmers refer to this operation as *single stepping* because it lets you step through the program one statement per operation. Ultimately, though, the real purpose of single stepping is to let you observe the results of a sequence of instructions, noting all side effects, so you can see why that sequence is not producing desired results.

CodeView provides two easy to use trace/single step commands. Pressing F8 *traces* through one instruction. CodeView will update all affected registers and memory locations and halt on the very next instruction. In the event the current instruction is a call, int, or other transfer of control instruction, CodeView transfers control to the target location and displays the instruction at that location.

The second CodeView command for single stepping is the *step* command. You can execute the step command by pressing F10. The step command executes the current statement and stops upon executing the statement immediately following it in the program. For most instructions the step and trace commands do the same thing. However, for instructions that transfer control, the trace command follows the flow of control while the step command allows the CPU to run at full speed until returning back to the next instruction. This, for example, lets you quickly execute a subroutine without having to step through all the instructions in that subroutine. You should attempt to using the program trace command (F8) for most debugging purposes and only use the step command (F10) on call and int instructions. The step instruction may have some unintended effects on other transfer of control instructions like loop, and the conditional branches.

The CodeView command window also provides two commands to trace or single step through an instruction. The "T" command traces through an instruction, the "P" command steps over an instruction.

One major problem with tracing through your program is that it is very slow. Even if you hold the F8 key down and let it autorepeat, you'd only be executing 10-20 instructions per second. This is a million (or more) times slower than a typical high-end PC. If the program executes several thousand instructions before even getting to the point where you

suspect the bug will be, you would have to execute far too many trace operations to get to that point.

A *breakpoint* is a point in your program where control returns to the debugger. This is the facility that lets you run a program a full speed up to a specific point (the break point) in your program. Breakpoints are, perhaps, the most important tool for locating errors in a machine language program. Since they are so useful, it is not surprising to find that CodeView provides a very rich set of breakpoint manipulation commands.

There are three keystroke commands that let you run your program at full speed and set breakpoints. The F5 command (run) begins full speed execution of your program at CS:IP. If you do not have any breakpoints set, your program will run to completion. If you are interested in stopping your program at some point you should set a breakpoint before executing this command.

Pressing F5 produces the same result as the "G" (go) command in the command window. The Go command is a little more powerful, however, because it lets you specify a *non-sticky breakpoint* at the same time. The command window Go commands take the following forms:

G

G *breakpoint_address*

The F7 keystroke executes at full speed up to the instruction the cursor is on. This sets a *non-sticky* breakpoint. To use this command you must first place the cursor on an instruction in the source window and then press the F7 key. CodeView will set a breakpoint at the specified instruction and start the program running at full speed until it hits a breakpoint.

A *non-sticky breakpoint* is one that deactivates whenever control returns back to CodeView. Once CodeView regains control it clears all non-sticky breakpoints. You will have to reset those breakpoints if you still need to stop at that point in your program. Note that CodeView clears the non-sticky breakpoints even if the program stops for some reason other than execution of those non-sticky breakpoints.

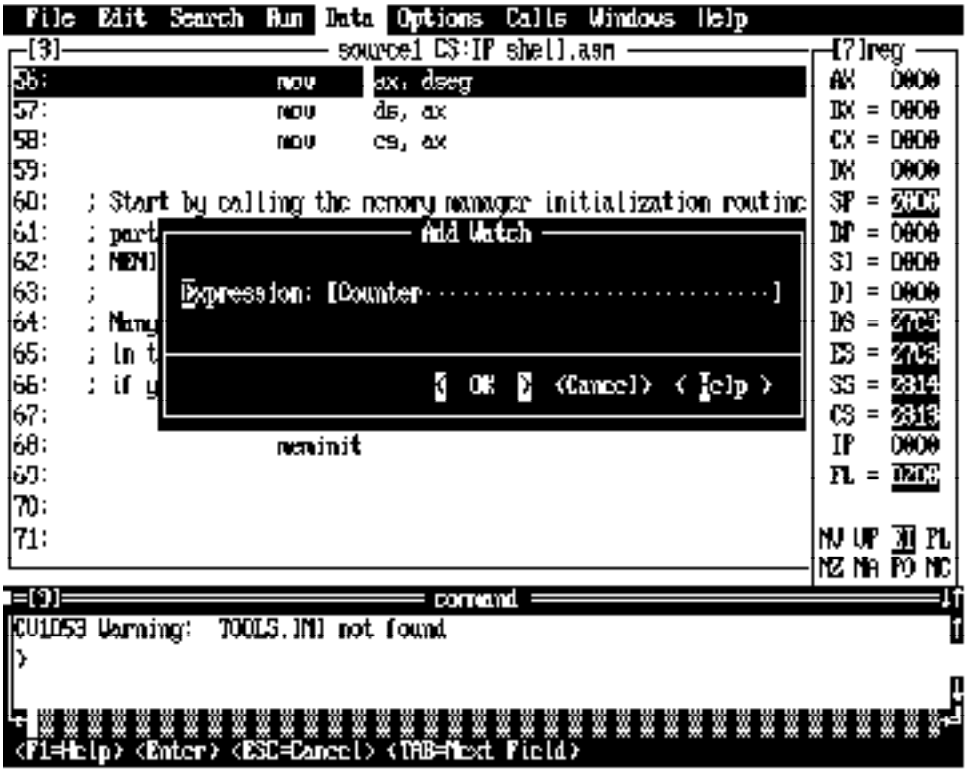
One very important thing to keep in mind, especially when using the F7 command to set non-sticky breakpoints, is that you must execute the statement on which the breakpoint was set for the breakpoint to have any effect. If your program skips over the instruction on which you've set the breakpoint, you might not return to CodeView except via program termination. When choosing a point for a breakpoint, you should always pick a *sequence point*. A sequence point is some spot in your program to which all execution paths converge. If you cannot set a breakpoint at a sequence point, you should set several breakpoints in your program if you are not sure the code will execute the statement with the single breakpoint.

The easiest way to set a sticky breakpoint is to move the cursor to the desired statement in the CodeView source window and press F9. This will *brighten* that statement to show that there is a breakpoint set on that instruction. Note that the F9 key only works on 80x86 machine instructions. You cannot use it on blank lines, comments, assembler directives, or pseudo-opcodes.

CodeView's command window also provides several commands to manipulate breakpoints including BC (Breakpoint Clear), BD (Breakpoint Disable), BE (Breakpoint Enable), BL (Breakpoint List), and BP (BreakPoint set). These commands are very powerful and let you set breakpoints on memory modification, expression evaluation, apply counters to breakpoints, and more. See the MASM "Environment and Tools" manual or the CodeView on-line help for more information about these commands.

Another useful debugging tool in CodeView is the *Watch Window*. The watch window displays the values of some specified expressions during program execution. One important use of the watch window is to display the contents of selected variables while your program executes. Upon encountering a breakpoint, CodeView automatically updates all

watch expressions. You can add a watch expression to the watch window using the DATA:Add Watch menu item. This opens up a dialog box that looks like the following:



By typing a variable name (like Counter above) you can add a watch item to the watch window. By opening the watch windows (from the Windows menu item) you can view the values of any watch expressions you've created.

Watch expressions are quite useful because they let you observe how your program affects the values of variables throughout your code. If you place several variable names in the watch list you can execute a section of code up to a break point and observe how that code affected certain variables.

9.10.2 Debugging Strategies

Learning how to effectively use a debugger to locate problems in your machine language programs is not something you can learn from a book. Alas, there is a bit of a learning curve to using a debugger like CodeView and learning the necessary techniques to quickly locate the source of an error within a program. For this reason all too many students fall back to debugging techniques they learned in their first or second quarter of programming, namely sticking a bunch of print statements throughout their code. You should not make this mistake. The time you spend learning how to properly use CodeView will pay off very quickly.

9.10.2.1 Locating Infinite Loops

Infinite loops are a very common problem in many programs. You start a program running and the whole machine locks up on you. How do you deal with this? Well, the first thing to do is to load your program into CodeView. Once you start your program running and it appears to be in an infinite loop, you can manually break the program by pressing the SysReq or Ctrl-Break key. This generally forces control back to CodeView. If

you are currently executing in a small loop, you can use the trace command to step through the loop and figure out why it does not terminate.

Another way to catch an infinite loop is to use a *binary search*. To use this technique, place a breakpoint in the middle of your program (or in the middle of the code you wish to test). Start the program running. If it hangs up, the infinite loop is *before* the breakpoint. If you execute the breakpoint, then the infinite loop occurs *after* the breakpoint[3] Once you determine which half of your program contains the infinite loop, the next step is to place another breakpoint half way into that part of the program. If the infinite loop occurred before the breakpoint in the middle of the program, then you should set a new breakpoint one quarter of the way into the program, that is, halfway between the beginning of the program and the original breakpoint. If you got to the original breakpoint without encountering the infinite loop, then set a new breakpoint at the three-quarters point in your program, i.e., halfway between the original breakpoint and the end of your program. Run the program from the beginning again (you can use the CodeView command window command "L" to restart the program from the beginning). If you do not hit any of the three breakpoints you know that the infinite loop is in the first 25% of the program. Otherwise, the current breakpoints at the 25%, 50%, and 75% points in the program will effectively limit the source of the infinite loop to a smaller section of your program. You can repeat this step over and over again until you pinpoint the section of your program containing the infinite loop.

Of course, you should not place a breakpoint within a loop when searching for an infinite loop. Otherwise CodeView will break on each iteration of the loop and it will take you much longer to find the error. Of course, if the infinite loop occurs *inside* some other loop you will eventually need to place breakpoints inside a loop, but hopefully you will find the infinite loop on the first execution of the outside loop. If you do need to place a breakpoint inside a loop that must execute several times before you really want the break to occur, you can attach a *counter* to a breakpoint that counts down from some value before actually breaking. See the MASM Environment and Tools manual, or use CodeView's on-line help facility, to get more details on breakpoint counters.

9.10.2.2 Incorrect Computations

Another common problem is that you get the wrong result after performing a sequence of arithmetic and logical computations. You can look at a section of code all day long and still not see the problem, but if you trace through the code, the incorrect code because quite obvious.

If you think that a particular computation is not producing a correct result you should set a breakpoint at the first instruction of the computation and run the program at full speed up to that point. *Be sure to check the values of all variables and registers used in the computation.* All too often a bad computation is the result of bad input values, that means the incorrect computation is elsewhere in your program.

Once you have verified that the input values are correct, you can being tracing the instructions of the computation one at a time. After each instruction executes you should compare the results you actually obtain against those you expected to obtain.

The main thing to keep in mind when trying to determine why your program is producing incorrect results is that the source of the error could be somewhere else besides the point where you first notice the error. This is why you should always check in input register and variable values before tracing through a section of code. If you find that the input values are *no* correct, then the problem lies elsewhere in your program and you will have to search elsewhere.

3. Of course, you must make sure that the instruction on which you set the break point is a sequence point. If the code can jump over your breakpoint into the second half of the program, you have proven nothing.

9.10.2.3 Illegal Instructions/Infinite Loops Part II

Sometimes when your program hangs up it is not due to the execution of an infinite loop, but rather you've executed an opcode that is not a valid machine instruction. Other times you will press the SysReq key only to find you are executing code that is nowhere near your program, perhaps out in the middle of RAM and executing some really weird instructions. Most of the time this is due to a stack problem or executing some indirect jump. The best strategy here is to open a memory window and dump some memory around the stack pointer (SS:SP). Try and locate a reasonable return address on the top of stack (or shortly thereafter if there are many values pushed on the stack) and disassemble that code. Somewhere before the return address is probably a call. You should set a breakpoint at that location and begin single stepping into the routine, watching what happens on all indirect jumps and returns. Pay close attention to the stack during all this.

9.10.3 Debug Exercise I: Using CodeView to Find Bugs in a Calculation

Exercise 1: Running CodeView. The following program contains several bugs (noted in the comments). Enter this program into the system (note, this code is available as the file Ex9_1.asm on the companion CD-ROM):

```
dseg            segment  para public 'data'

I               word     0
J               word     0
K               word     0

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; This program is useful for debugging purposes only!
; The intent is to execute this code from inside CodeView.
;
; This program is riddled with bugs. The bugs are very
; obvious in this short code sequence, within a larger
; program these bugs might not be quite so obvious.

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax

; The following loop increments I until it reaches 10

ForILoop:       inc      I
                cmp      I, 10
                jb       ForILoop

; This loop is supposed to do the same thing as the loop
; above, but we forgot to reinitialize I back to zero.
; What happens?

ForILoop2:      inc      I
                cmp      I, 10
                jb       ForILoop2

; The following loop, once again, attempts to do the same
; thing as the first for loop above. However, this time we
; remembered to reinitialize I. Alas, there is another
; problem with this code, a typo that the assembler cannot
; catch.

                mov      I, 0
ForILoop3:      inc      I
                cmp      I, 10
                jb       ForILoop    ;<<<-- Whoops! Typo.
```

```
; The following loop adds I to J until J reaches 100.
; Unfortunately, the author of this code must have been
; confused and thought that AX contained the sum
; accumulating in J. It compares AX against 100 when
; it should really be comparing J against 100.

WhileJLoop:     mov     ax, I
                add     J, ax
                cmp     ax, 100     ;This is a bug!
                jb      WhileJLoop

                mov     ah, 4ch     ;Quit to DOS.
                int     21h
Main            endp
cseg ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

Assemble this program with the command:

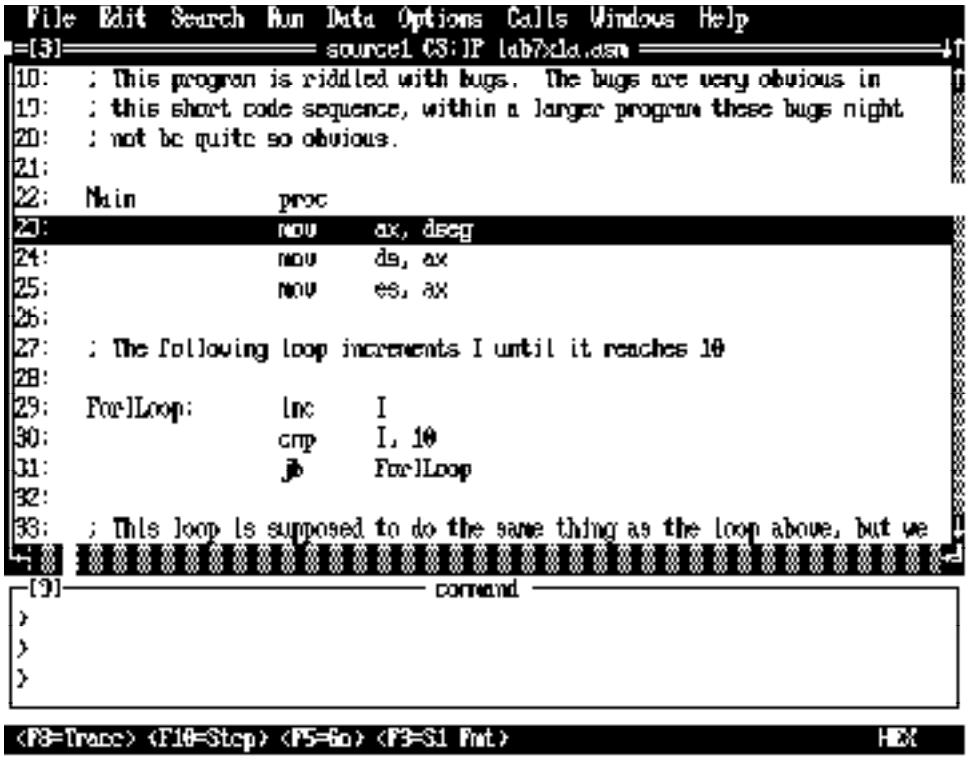
```
ML /Zi Ex9_1.asm
```

The "/Zi" option instructions MASM to include debugging information for CodeView in the .EXE file. Note that the "Z" must be uppercase and the "i" must be lower case.

Load this into CodeView using the command:

```
CV Ex9_1
```

Your display should now look something like the following:



Note that CodeView highlights the instruction it will execute next (mov ax, dseg in the above code). Try out the trace command by pressing the F10 key three times.This should leave the inc I instruction highlighted. Step through the loop and note all the major

changes that take place on each iteration (note: remember jb=jc so be sure to note the value of the carry flag on each iteration as well).

For your lab report: Discuss the results in your lab manual. Also note the final value of I after completing the loop.

Part Two: Locating a bug. The second loop in the program contains a major bug. The programmer forgot to reset I back to zero before executing the code starting at label ForlLoop2. Trace through this loop until it falls through to the statement at label ForlLoop3.

For your lab report: Describe what went wrong and how pressing the F8 key would help you locate this problem.

Part 3: Locating another bug. The third loop contains a typo that causes it to restart at label ForlLoop. Trace through this code using the F8 key.

For your lab report: Describe the process of tracking this problem down and provide a description of how you could use the trace command to catch this sort of problem.

Part 4: Verifying correctness. Program Ex9_2.asm is a corrected version of the above program. Single step through that code and verify that it works correctly.

For your lab report: Describe the differences between the two debugging sessions in your lab manual.

Part 5: Using Ex9_2.asm, open a watch window and add the watch expression “I” to that window. Set sticky breakpoints on the three jb instructions in the program. Run the program using the Go command and comment on what happens in the Watch window at each breakpoint.

For your lab report: Describe how you could use the watch window to help you locate a problem in your programs.

9.10.4 Software Delay Loop Exercises

Software Delay Loops. The Ex9_3.asm file contains a short software-based delay loop. Run this program and determine the value for the loop control variable that will cause a delay of 11 seconds. Note: the current value was chosen for a 66 MHz 80486 system; if you have a slower system you may want to reduce this value, if you have a faster system, you will want to increase this value. Adjust the value to get the delay as close to 11 seconds as you can on your PC.

For your lab report: Provide the constant for you particular system that produces a delay of 11 seconds. Discuss how to create a delay of 1, 10, 20, 30, or 60 seconds using this code.

For additional credit: After getting the delay loop to run for 11 seconds on your PC, take the executable around to different systems with different CPUs and different clock speeds. Run the program and measure the delay. Describe the differences in your lab report.

Part 2: Hardware determined software delay loop. The Ex9_4.asm file contains a software delay loop that automatically determines the number of loop iterations by observing the BIOS real time clock variable. Run this software and observe the results.

For your lab report: Determine the loop iteration count and include this value in your lab manual. If your PC has a turbo switch on it, set it to “non-turbo” mode when requested by the program. Measure the actual delay as accurately as you can with the turbo switch in turbo and in non-turbo mode. Include these timings in your lab report.

For additional credit: Take the executable file around to different systems with different CPUs and different clock speeds. Run the program and measure the delays. Describe the differences in your lab report.

9.11 Programming Projects

9.12 Summary

This chapter discussed arithmetic and logical operations on 80x86 CPUs. It presented the instructions and techniques necessary to perform integer arithmetic in a fashion similar to high level languages. This chapter also discussed multiprecision operations, how to perform arithmetic operations using non-arithmetic instructions, and how to use arithmetic instructions to perform non-arithmetic operations.

Arithmetic expressions are much simpler in a high-level language than in assembly language. Indeed, the original purpose of the FORTRAN programming language was to provide a FORMula TRANslator for arithmetic expressions. Although it takes a little more effort to convert an arithmetic formula to assembly language than it does to, say, Pascal, as long as you follow some very simple rules the conversion is not hard. For a step-by-step description, see

- “Arithmetic Expressions” on page 460
- “Simple Assignments” on page 460
- “Simple Expressions” on page 460
- “Complex Expressions” on page 462
- “Commutative Operators” on page 466
- “Logical (Boolean) Expressions” on page 467

One big advantage to assembly language is that it is easy to perform nearly unlimited precision arithmetic and logical operations. This chapter describes how to do extended precision operations for most of the common operations. For complete instructions, see

- “Multiprecision Operations” on page 470
- “Multiprecision Addition Operations” on page 470
- “Multiprecision Subtraction Operations” on page 472
- “Extended Precision Comparisons” on page 473
- “Extended Precision Multiplication” on page 475
- “Extended Precision Division” on page 477
- “Extended Precision NEG Operations” on page 480
- “Extended Precision AND Operations” on page 481
- “Extended Precision OR Operations” on page 482
- “Extended Precision NOT Operations” on page 482
- “Extended Precision Shift Operations” on page 482
- “Extended Precision Rotate Operations” on page 484

At certain times you may need to operate on two operands that are different types. For example, you may need to add a byte value to a word value. The general idea is to extend the smaller operand so that it is the same size as the larger operand and then compute the result on these like-sized operands. For all the details, see

- “Operating on Different Sized Operands” on page 485

Although the 80x86 instruction set provides straight-forward ways to accomplish many tasks, you can often take advantage of various idioms in the instruction set or with respect to certain arithmetic operations to produce code that is faster or shorter than the obvious way. This chapter introduces a few of these idioms. To see some examples, check out

- “Machine and Arithmetic Idioms” on page 486
- “Multiplying Without MUL and IMUL” on page 487
- “Division Without DIV and IDIV” on page 488
- “Using AND to Compute Remainders” on page 488
- “Implementing Modulo-n Counters with AND” on page 489
- “Testing an Extended Precision Value for 0FFFF..FFh” on page 489

- “TEST Operations” on page 489
- “Testing Signs with the XOR Instruction” on page 490

To manipulate packed data you need the ability to extract a field from a packed record and insert a field into a packed record. You can use the logical and and or instructions to mask the fields you want to manipulate; you can use the shl and shr instructions to position the data to their appropriate positions before inserting or after extracting data. To learn how to pack and unpack data, see

- “Masking Operations” on page 490
- “Masking Operations with the AND Instruction” on page 490
- “Masking Operations with the OR Instruction” on page 491
- “Packing and Unpacking Data Types” on page 491

9.13 Questions

1) Describe how you might go about adding an unsigned word to an unsigned byte variable producing a byte result. Explain any error conditions and how to check for them.

2) Answer question one for signed values.

3) Assume that var1 is a word and var2 and var3 are double words. What is the 80x86 assembly language code that will add var1 to var2 leaving the sum in var3 if:

   a) var1, var2, and var3 are unsigned values.

   b) var1, var2, and var3 are signed values.

4) "ADD BX, 4" is more efficient than "LEA BX, 4[BX]". Give an example of an LEA instruction which is more efficient than the corresponding ADD instruction.

5) Provide the single 80386 LEA instruction that will multiply EAX by five.

6) Assume that VAR1 and VAR2 are 32 bit variables declared with the DWORD pseudo-opcode. Write code sequences that will test the following:

   a) VAR1 = VAR2

   b) VAR1 <> VAR2

   c) VAR1 < VAR2 (Unsigned and signed versions

   d) VAR1 <= VAR2 for each of these)

   e) VAR1 > VAR2

   f) VAR1 >= VAR2

7) Convert the following expressions into assembly language code employing shifts, additions, and subtractions in place of the multiplication:

   a) AX*15

   b) AX*129

   c) AX*1024

   d) AX*20000

8) What's the best way to divide the AX register by the following constants?

   a) 8 b) 255 c) 1024 d) 45

9) Describe how you could multiply an eight bit value in AL by 256 (leaving the result in AX) using nothing more than two MOV instructions.

10) How could you logically AND the value in AX by 0FFh using nothing more than a MOV instruction?

11) Suppose that the AX register contains a pair of packed binary values with the L.O. four bits containing a value in the range 0..15 and the H.O. 12 bits containing a value in the range 0..4095. Now suppose you want to see if the 12 bit portion contains the value 295. Explain how you could accomplish this with two instructions.

12) How could you use the TEST instruction (or a sequence of TEST instructions) to see if bits zero and four in the AL register are both set to one? How would the TEST instruction be used to see if either bit is set? How could the TEST instruction be used to see if neither bit is set?

13) Why can't the CL register be used as a count operand when shifting multi-precision operands. I.e., why won't the following instructions shift the value in (DX,AX) three bits to the left?

```
mov         cl, 3
shl         ax, cl
rcl         dx, cl
```

14) Provide instruction sequences that perform an extended precision (32 bit) ROL and ROR operation using only 8086 instructions.

15) Provide an instruction sequence that implements a 64 bit ROR operation using the 80386 SHRD and BT instructions.

16) Provide the 80386 code to perform the following 64 bit computations. Assume you are computing X := Y op Z with X, Y, and Z defined as follows:

```
X               dword   0, 0
y               dword   1, 2
z               dword   3, 4
```

a) addition b) subtraction c) multiplication

c) Logical AND d) Logical OR e) Logical XOR

f) negate g) Logical NOT

Control Structures Chapter 10

A computer program typically contains three structures: instruction sequences, decisions, and loops. A sequence is a set of sequentially executing instructions. A decision is a branch (goto) within a program based upon some condition. A loop is a sequence of instructions that will be repeatedly executed based on some condition. In this chapter we will explore some of the common decision structures in 80x86 assembly language.

10.0 Chapter Overview

This chapter discusses the two primary types of control structures: decision and iteration. It describes how to convert high level language statements like if..then..else, case (switch), while, for etc., into equivalent assembly language sequences. This chapter also discusses techniques you can use to improve the performance of these control structures. The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- Introduction to Decisions.
- IF..THEN..ELSE Sequences.
- CASE Statements.
- ❑ State machines and indirect jumps.
- Spaghetti code.
- Loops.
- WHILE Loops.
- REPEAT..UNTIL loops.
- LOOP..ENDLOOP.
- FOR Loops.
- Register usage and loops.
- ❑ Performance improvements.
- ❑ Moving the termination condition to the end of a loop.
- ❑ Executing the loop backwards.
- ❑ Loop invariants.
- ❑ Unraveling loops.
- ❑ Induction variables.

10.1 Introduction to Decisions

In its most basic form, a decision is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the if..then..else statement in Pascal:

```
IF (condition is true) THEN stmt1 ELSE stmt2 ;
```

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following Pascal statement:

```
IF ((X<Y) and (Z > T)) or (A <> B) THEN stmt1;
```

A "brute force" approach to converting this statement into assembly language might produce:

```
                mov     cl, 1       ;Assume true
                mov     ax, X
                cmp     ax, Y
                jl      IsTrue
                mov     cl, 0       ;This one's false
IsTrue:         mov     ax, Z
                cmp     ax, T
                jg      AndTrue
                mov     cl, 0       ;It's false now
AndTrue:        mov     al, A
                cmp     al, B
                je      OrFalse
                mov     cl, 1       ;Its true if A <> B
OrFalse:        cmp     cl, 1
                jne     SkipStmt1
        <Code for stmt1 goes here>
SkipStmt1:
```

As you can see, it takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) Pascal statements:

```
                cl := true;
                IF (X >= Y) then cl := false;
                IF (Z <= T) then cl := false;
                IF (A <> B) THEN cl := true;
                IF (CL = true) then stmt1;
```

Now compare this with the following "improved" code:

```
                mov     ax, A
                cmp     ax, B
                jne     DoStmt
                mov     ax, X
                cmp     ax, Y
                jnl     SkipStmt
                mov     ax, Z
                cmp     ax, T
                jng     SkipStmt
DoStmt:
        <Place code for Stmt1 here>

SkipStmt:
```

Two things should be apparent from the code sequences above: first, a single conditional statement in Pascal may require several conditional jumps in assembly language; second, organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, care should be exercised when dealing with conditional sequences in assembly language.

Conditional statements may be broken down into three basic categories: if..then..else statements, case statements, and indirect jumps. The following sections will describe these program structures, how to use them, and how to write them in assembly language.

10.2 IF..THEN..ELSE Sequences

The most commonly used conditional statement is theif..then or if..then..else statement. These two statements take the following form shown in Figure 10.1.

The if..then statement is just a special case of the if..then..else statement (with an empty ELSE block). Therefore, we'll only consider the more general if..then..else form. The basic implementation of an if..then..else statement in 80x86 assembly language looks something like this:

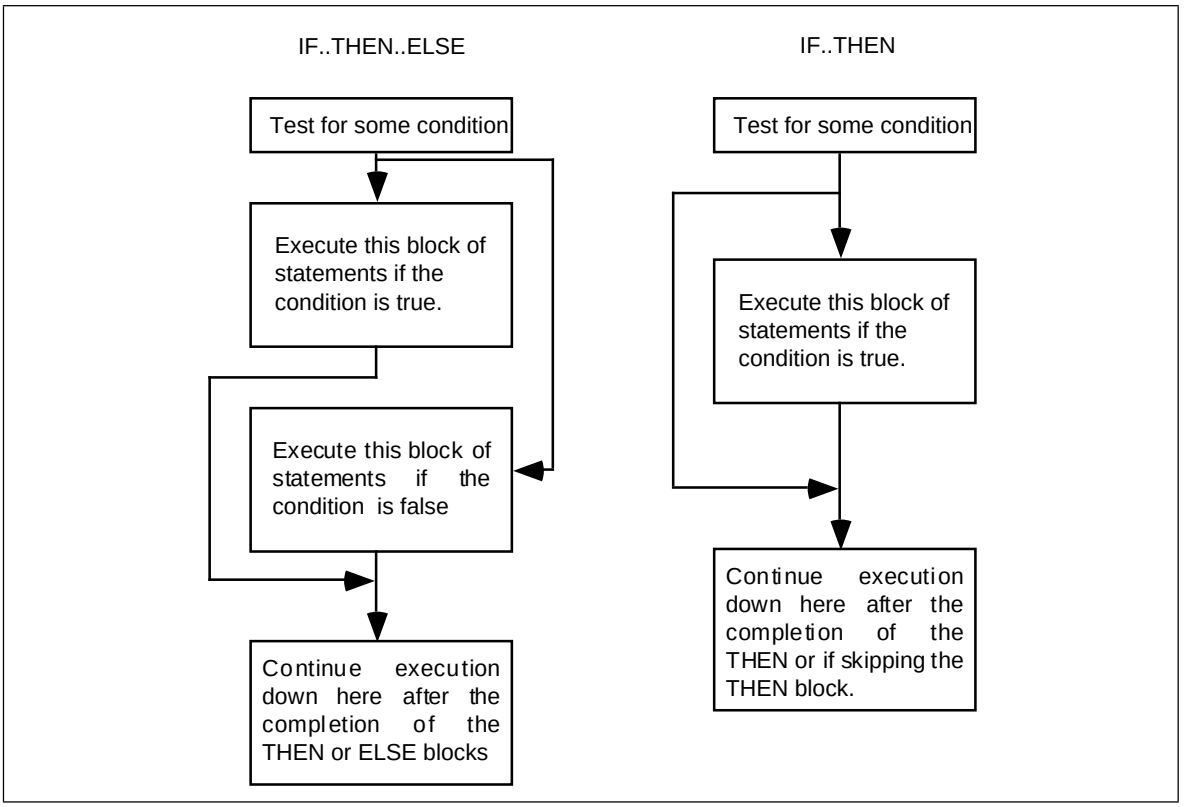


Figure 10.1 IF..THEN and IF..THEN..ELSE Statement Flow

```
        {Sequence of statements to test some condition}
                Jcc     ElseCode
        {Sequence of statements corresponding to the THEN block}
                jmp     EndOfIF
ElseCode:
        {Sequence of statements corresponding to the ELSE block}
EndOfIF:
```

Note: J*cc* represents some conditional jump instruction.

For example, to convert the Pascal statement:

```
IF (a=b) then c := d else b := b + 1;
```

to assembly language, you could use the following 80x86 code:

```
                mov     ax, a
                cmp     ax, b
                jne     ElseBlk
                mov     ax, d
                mov     c, ax
                jmp     EndOfIf
ElseBlk:
                inc     b
EndOfIf:
```

For simple expressions like (A=B) generating the proper code for an if..then..else statement is almost trivial. Should the expression become more complex, the associated assembly language code complexity increases as well. Consider the following if statement presented earlier:

```
IF ((X > Y) and (Z < T)) or (A<>B) THEN C := D;
```

When processing complex if statements such as this one, you'll find the conversion task easier if you break this if statement into a sequence of three different if statements as follows:

```
IF (A<>B) THEN C := D
IF (X > Y) THEN   IF (Z < T) THEN C := D;
```

This conversion comes from the following Pascal equivalences:

```
IF (expr1 AND expr2) THEN stmt;
```

is equivalent to

```
IF (expr1) THEN IF (expr2) THEN stmt;
```

and

```
IF (expr1 OR expr2) THEN stmt;
```

is equivalent to

```
IF (expr1) THEN stmt;
IF (expr2) THEN stmt;
```

In assembly language, the former if statement becomes:

```
                mov     ax, A
                cmp     ax, B
                jne     DoIF
                mov     ax, X
                cmp     ax, Y
                jng     EndOfIf
                mov     ax, Z
                cmp     ax, T
                jnl     EndOfIf
DoIf:
                mov     ax, D
                mov     C, ax
EndOfIF:
```

As you can probably tell, the code necessary to test a condition can easily become more complex than the statements appearing in the else and then blocks. Although it seems somewhat paradoxical that it may take more effort to test a condition than to act upon the results of that condition, it happens all the time. Therefore, you should be prepared for this situation.

Probably the biggest problem with the implementation of complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. Probably the biggest advantage high level languages offer over assembly language is that expressions are much easier to read and comprehend in a high level language. The HLL version is self-documenting whereas assembly language tends to hide the true nature of the code. Therefore, well-written comments are an essential ingredient to assembly language implementations of if..then..else statements. An elegant implementation of the example above is:

```
; IF ((X > Y) AND (Z < T)) OR (A <> B) THEN C := D;
; Implemented as:
; IF (A <> B) THEN GOTO DoIF;

                mov     ax, A
                cmp     ax, B
                jne     DoIF

; IF NOT (X > Y) THEN GOTO EndOfIF;

                mov     ax, X
                cmp     ax, Y
                jng     EndOfIf

; IF NOT (Z < T) THEN GOTO EndOfIF ;

                mov     ax, Z
                cmp     ax, T
                jnl     EndOfIf
```

```
; THEN Block:

DoIf:           mov     ax, D
                mov     C, ax

; End of IF statement

EndOfIF:
```

Admittedly, this appears to be going overboard for such a simple example. The following would probably suffice:

```
; IF ((X > Y) AND (Z < T)) OR (A <> B) THEN C := D;

; Test the boolean expression:

                mov     ax, A
                cmp     ax, B
                jne     DoIF
                mov     ax, X
                cmp     ax, Y
                jng     EndOfIf
                mov     ax, Z
                cmp     ax, T
                jnl     EndOfIf

; THEN Block:

DoIf:           mov     ax, D
                mov     C, ax

; End of IF statement

EndOfIF:
```

However, as your if statements become complex, the density (and quality) of your comments become more and more important.

10.3 CASE Statements

The Pascal case statement takes the following form :

```
CASE variable OF
          const1:stmt1;
          const2:stmt2;
           .
           .
           .
          constn:stmtn
END;
```

When this statement executes, it checks the value of variable against the constants $const_1$ ... $const_n$. If a match is found then the corresponding statement executes. Standard Pascal places a few restrictions on the case statement. First, if the value of variable isn't in the list of constants, the result of the case statement is undefined. Second, all the constants appearing as case labels must be unique. The reason for these restrictions will become clear in a moment.

Most introductory programming texts introduce the case statement by explaining it as a sequence of if..then..else statements. They might claim that the following two pieces of Pascal code are equivalent:

```
CASE I OF
          0: WriteLn('I=0');
          1: WriteLn('I=1');
          2: WriteLn('I=2');
END;

IF I = 0 THEN WriteLn('I=0')
ELSE IF I = 1 THEN WriteLn('I=1')
ELSE IF I = 2 THEN WriteLn('I=2');
```

While semantically these two code segments may be the same, their implementation is usually different[1]. Whereas the if..then..else if chain does a comparison for each conditional statement in the sequence, the case statement normally uses an indirect jump to transfer control to any one of several statements with a single computation. Consider the two examples presented above, they could be written in assembly language with the following code:

```
                mov     bx, I
                shl     bx, 1       ;Multiply BX by two
                jmp     cs:JmpTbl[bx]

JmpTbl          word    stmt0, stmt1, stmt2

Stmt0:          print
                byte    "I=0",cr,lf,0
                jmp     EndCase

Stmt1:          print
                byte    "I=1",cr,lf,0
                jmp     EndCase

Stmt2:          print
                byte    "I=2",cr,lf,0

EndCase:

; IF..THEN..ELSE form:

                mov     ax, I
                cmp     ax, 0
                jne     Not0
                print
                byte    "I=0",cr,lf,0
                jmp     EndOfIF

Not0:           cmp     ax, 1
                jne     Not1
                print
                byte    "I=1",cr,lf,0
                jmp     EndOfIF

Not1:           cmp     ax, 2
                jne     EndOfIF
                Print
                byte    "I=2",cr,lf,0
EndOfIF:
```

Two things should become readily apparent: the more (consecutive) cases you have, the more efficient the jump table implementation becomes (both in terms of space and speed). Except for trivial cases, the case statement is almost always faster and usually by a large margin. As long as the case labels are consecutive values, the case statement version is usually smaller as well.

What happens if you need to include non-consecutive case labels or you cannot be sure that the case variable doesn't go out of range? Many Pascals have extended the definition of the case statement to include an otherwise clause. Such a case statement takes the following form:

```
CASE variable OF
          const:stmt;
          const:stmt;
           . .
           . .
           . .
          const:stmt;
          OTHERWISE stmt
END;
```

If the value of variable matches one of the constants making up the case labels, then the associated statement executes. If the variable's value doesn't match any of the case

1. Versions of Turbo Pascal, sadly, treat the case statement as a form of the if..then..else statement.

labels, then the statement following the otherwise clause executes. The otherwise clause is implemented in two phases. First, you must choose the minimum and maximum values that appear in a case statement. In the following case statement, the smallest case label is five, the largest is 15:

```
CASE I OF
          5:stmt1;
          8:stmt2;
          10:stmt3;
          12:stmt4;
          15:stmt5;
          OTHERWISE stmt6
END;
```

Before executing the jump through the jump table, the 80x86 implementation of this case statement should check the case variable to make sure it's in the range 5..15. If not, control should be immediately transferred to stmt6:

```
                mov     bx, I
                cmp     bx, 5
                jl      Otherwise
                cmp     bx, 15
                jg      Otherwise
                shl     bx, 1
                jmp     cs:JmpTbl-10[bx]
```

The only problem with this form of the case statement as it now stands is that it doesn't properly handle the situation where I is equal to 6, 7, 9, 11, 13, or 14. Rather than sticking extra code in front of the conditional jump, you can stick extra entries in the jump table as follows:

```
                mov     bx, I
                cmp     bx, 5
                jl      Otherwise
                cmp     bx, 15
                jg      Otherwise
                shl     bx, 1
                jmp     cs:JmpTbl-10[bx]

Otherwise:      {put stmt6 here}
                jmp     CaseDone

JmpTbl          word    stmt1, Otherwise, Otherwise, stmt2, Otherwise
                word    stmt3, Otherwise, stmt4, Otherwise, Otherwise
                word    stmt5
                etc.
```

Note that the value 10 is subtracted from the address of the jump table. The first entry in the table is always at offset zero while the smallest value used to index into the table is five (which is multiplied by two to produce 10). The entries for 6, 7, 9, 11, 13, and 14 all point at the code for the Otherwise clause, so if I contains one of these values, the Otherwise clause will be executed.

There is a problem with this implementation of the case statement. If the case labels contain non-consecutive entries that are widely spaced, the following case statement would generate an extremely large code file:

```
CASE I OF
          0: stmt1;
          100: stmt2;
          1000: stmt3;
          10000: stmt4;
          Otherwise stmt5
END;
```

In this situation, your program will be much smaller if you implement the case statement with a sequence of if statements rather than using a jump statement. However, keep one thing in mind- the size of the jump table does not normally affect the execution speed of the program. If the jump table contains two entries or two thousand, the case statement will execute the multi-way branch in a constant amount of time. The if statement imple-

mentation requires a linearly increasing amount of time for each case label appearing in the case statement.

Probably the biggest advantage to using assembly language over a HLL like Pascal is that you get to choose the actual implementation. In some instances you can implement a case statement as a sequence ofif..then..else statements, or you can implement it as a jump table, or you can use a hybrid of the two:

```
CASE I OF
          0:stmt1;
          1:stmt2;
          2:stmt3;
          100:stmt4;
          Otherwise stmt5
END;
```

could become:

```
          mov       bx, I
          cmp       bx, 100
          je        Is100
          cmp       bx, 2
          ja        Otherwise
          shl       bx, 1
          jmp       cs:JmpTbl[bx]
          etc.
```

Of course, you could do this in Pascal with the following code:

```
IF I = 100 then stmt4
ELSE CASE I OF
          0:stmt1;
          1:stmt2;
          2:stmt3;
          Otherwise stmt5
END;
```

But this tends to destroy the readability of the Pascal program. On the other hand, the extra code to test for 100 in the assembly language code doesn't adversely affect the readability of the program (perhaps because it's so hard to read already). Therefore, most people will add the extra code to make their program more efficient.

The C/C++ switch statement is very similar to the Pascal case statement. There is only one major semantic difference: the programmer must explicitly place a break statement in each case clause to transfer control to the first statement beyond the switch. This break corresponds to the jmp instruction at the end of each case sequence in the assembly code above. If the corresponding break is not present, C/C++ transfers control into the code of the following case. This is equivalent to leaving off the jmp at the end of the case's sequence:

```
switch (i)
{
case 0:   stmt1;
case 1:   stmt2;
case 2:   stmt3;
          break;
case 3:   stmt4;
          break;
default:  stmt5;
}
```

This translates into the following 80x86 code:

```
          mov       bx, i
          cmp       bx, 3
          ja        DefaultCase

          shl       bx, 1
          jmp       cs:JmpTbl[bx]
JmpTbl    word      case0, case1, case2, case3
```

```
case0:          <stmt1's code>

case1:          <stmt2's code>

case2:          <stmt3's code>

                jmp     EndCase     ;Emitted for the break stmt.

case3:          <stmt4's code>
                jmp     EndCase     ;Emitted for the break stmt.

DefaultCase:    <stmt5's code>
EndCase:
```

10.4 State Machines and Indirect Jumps

Another control structure commonly found in assembly language programs is the *state machine*. A state machine uses a *state variable* to control program flow. The FORTRAN programming language provides this capability with the assigned goto statement. Certain variants of C (e.g., GNU's GCC from the Free Software Foundation) provide similar features. In assembly language, the indirect jump provides a mechanism to easily implement state machines.

So what is a state machine? In very basic terms, it is a piece of code[2] which keeps track of its execution history by entering and leaving certain "states". For the purposes of this chapter, we'll not use a very formal definition of a state machine. We'll just assume that a state machine is a piece of code which (somehow) remembers the history of its execution (its *state*) and executes sections of code based upon that history.

In a very real sense, all programs are state machines. The CPU registers and values in memory constitute the "state" of that machine. However, we'll use a much more constrained view. Indeed, for most purposes only a single variable (or the value in the IP register) will denote the current state.

Now let's consider a concrete example. Suppose you have a procedure which you want to perform one operation the first time you call it, a different operation the second time you call it, yet something else the third time you call it, and then something new again on the fourth call. After the fourth call it repeats these four different operations in order. For example, suppose you want the procedure to add ax and bx the first time, subtract them on the second call, multiply them on the third, and divide them on the fourth. You could implement this procedure as follows:

```
State           byte    0
StateMach       proc
                cmp     state,0
                jne     TryState1

; If this is state 0, add BX to AX and switch to state 1:

                add     ax, bx
                inc     State                   ;Set it to state 1
                ret

; If this is state 1, subtract BX from AX and switch to state 2

TryState1:      cmp     State, 1
                jne     TryState2
                sub     ax, bx
                inc     State
                ret

; If this is state 2, multiply AX and BX and switch to state 3:

TryState2:      cmp     State, 2
```

2. Note that state machines need not be software based. Many state machines' implementation are hardware based.

```
                jne     MustBeState3
                push    dx
                mul     bx
                pop     dx
                inc     State
                ret

; If none of the above, assume we're in State 4. So divide
; AX by BX.

MustBeState3:
                push    dx
                xor     dx, dx      ;Zero extend AX into DX.
                div     bx
                pop     dx
                mov     State, 0    ;Switch back to State 0
                ret
StateMach       endp
```

Technically, this procedure is not the state machine. Instead, it is the variable State and the cmp/jne instructions which constitute the state machine.

There is nothing particularly special about this code. It's little more than a case statement implemented via the if..then..else construct. The only thing special about this procedure is that it remembers how many times it has been called[3] and behaves differently depending upon the number of calls. While this is a *correct* implementation of the desired state machine, it is not particularly efficient. The more common implementation of a state machine in assembly language is to use an *indirect jump*. Rather than having a state variable which contains a value like zero, one, two, or three, we could load the state variable with the *address* of the code to execute upon entry into the procedure. By simply jumping to that address, the state machine could save the tests above needed to execute the proper code fragment. Consider the following implementation using the indirect jump:

```
State           word    State0
StateMach       proc
                jmp     State

; If this is state 0, add BX to AX and switch to state 1:

State0:         add     ax, bx
                mov     State, offset State1        ;Set it to state 1
                ret

; If this is state 1, subtract BX from AX and switch to state 2

State1:         sub     ax, bx
                mov     State, offset State2        ;Switch to State 2
                ret

; If this is state 2, multiply AX and BX and switch to state 3:

State2:         push    dx
                mul     bx
                pop     dx
                mov     State, offset State3        ;Switch to State 3
                ret

; If in State 3, do the division and switch back to State 0:

State3:         push    dx
                xor     dx, dx      ;Zero extend AX into DX.
                div     bx
                pop     dx
                mov     State, offset State0        ;Switch to State 0
                ret
StateMach       endp
```

The jmp instruction at the beginning of the StateMach procedure transfers control to the location pointed at by the State variable. The first time you call StateMach it points at

3. Actually, it remembers how many times, *MOD 4*, that it has been called.

the State0 label. Thereafter, each subsection of code sets the State variable to point at the appropriate successor code.

10.5 Spaghetti Code

One major problem with assembly language is that it takes several statements to realize a simple idea encapsulated by a single HLL statement. All too often an assembly language programmer will notice that s/he can save a few bytes or cycles by jumping into the middle of some programming structure. After a few such observations (and corresponding modifications) the code contains a whole sequence of jumps in and out of portions of the code. If you were to draw a line from each jump to its destination, the resulting listing would end up looking like someone dumped a bowl of spaghetti on your code, hence the term "spaghetti code".

Spaghetti code suffers from one major drawback- it's difficult (at best) to read such a program and figure out what it does. Most programs start out in a "structured" form only to become spaghetti code at the altar of efficiency. Alas, spaghetti code is rarely efficient. Since it's difficult to figure out exactly what's going on, it's very difficult to determine if you can use a better algorithm to improve the system. Hence, spaghetti code may wind up less efficient.

While it's true that producing some spaghetti code in your programs may improve its efficiency, doing so should always be a last resort (when you've tried everything else and you still haven't achieved what you need), never a matter of course. Always start out writing your programs with straight-forward ifs and case statements. Start combining sections of code (via jmp instructions) once everything is working and well understood. Of course, you should never obliterate the structure of your code unless the gains are worth it.

A famous saying in structured programming circles is "After gotos, pointers are the next most dangerous element in a programming language." A similar saying is "Pointers are to data structures what gotos are to control structures." In other words, avoid excessive use of pointers. If pointers and gotos are bad, then the indirect jump must be the worst construct of all since it involves both gotos and pointers! Seriously though, the indirect jump instructions should be avoided for casual use. They tend to make a program harder to read. After all, an indirect jump can (theoretically) transfer control to any label within a program. Imagine how hard it would be to follow the flow through a program if you have no idea what a pointer contains and you come across an indirect jump using that pointer. Therefore, you should always exercise care when using jump indirect instructions.

10.6 Loops

Loops represent the final basic control structure (sequences, decisions, and loops) which make up a typical program. Like so many other structures in assembly language, you'll find yourself using loops in places you've never dreamed of using loops. Most HLLs have implied loop structures hidden away. For example, consider the BASIC statement IF A$ = B$ THEN 100. This if statement compares two strings and jumps to statement 100 if they are equal. In assembly language, you would need to write a loop to compare each character in A$ to the corresponding character in B$ and then jump to statement 100 if and only if all the characters matched. In BASIC, there is no loop to be seen in the program. In assembly language, this very simple if statement requires a loop. This is but a small example which shows how loops seem to pop up everywhere.

Program loops consist of three components: an optional initialization component, a loop termination test, and the body of the loop. The order with which these components are assembled can dramatically change the way the loop operates. Three permutations of these components appear over and over again. Because of their frequency, these loop structures are given special names in HLLs: while loops, repeat..until loops (do..while in C/C++), and loop..endloop loops.

10.6.1 While Loops

The most general loop is the while loop. It takes the following form:

```
WHILE boolean expression DO statement;
```

There are two important points to note about the while loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of the position of the termination test, the body of the loop may never execute. If the termination condition always exists, the loop body will always be skipped over.

Consider the following Pascal while loop:

```
I := 0;
WHILE (I<100) do I := I + 1;
```

I := 0; is the initialization code for this loop. I is a loop control variable, because it controls the execution of the body of the loop. (I<100) is the loop termination condition. That is, the loop will not terminate as long as I is less than 100. I:=I+1; is the loop body. This is the code that executes on each pass of the loop. You can convert this to 80x86 assembly language as follows:

```
                mov     I, 0
WhileLp:        cmp     I, 100
                jge     WhileDone
                inc     I
                jmp     WhileLp

WhileDone:
```

Note that a Pascal while loop can be easily synthesized using an if and a goto statement. For example, the Pascal while loop presented above can be replaced by:

```
        I := 0;
        1:      IF (I<100) THEN BEGIN
                I := I + 1;
                GOTO 1;
        END;
```

More generally, any while loop can be built up from the following:

```
        optional initialization code
1:      IF not termination condition THEN BEGIN
                loop body
                GOTO 1;
        END;
```

Therefore, you can use the techniques from earlier in this chapter to convert if statements to assembly language. All you'll need is an additional jmp (goto) instruction.

10.6.2 Repeat..Until Loops

The repeat..until (do..while) loop tests for the termination condition at the end of the loop rather than at the beginning. In Pascal, the repeat..until loop takes the following form:

```
optional initialization code
REPEAT
        loop body
UNTIL termination condition
```

This sequence executes the initialization code, the loop body, then tests some condition to see if the loop should be repeated. If the boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things to note about the repeat..until loop is that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body executes at least once.

Like the while loop, the repeat..until loop can be synthesized with an if statement and a goto . You would use the following:

```
        initialization code
1:              loop body
        IF NOT termination condition THEN GOTO 1
```

Based on the material presented in the previous sections, you can easily synthesize repeat..until loops in assembly language.

10.6.3 LOOP..ENDLOOP Loops

If while loops test for termination at the beginning of the loop and repeat..until loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. Although Pascal and C/C++[4] don't directly support such a loop, the loop..endloop structure can be found in HLL languages like Ada. The loop..endloop loop takes the following form:

```
        LOOP
                loop body
        ENDLOOP;
```

Note that there is no explicit termination condition. Unless otherwise provided for, the loop..endloop construct simply forms an infinite loop. Loop termination is handled by an if and goto statement[5]. Consider the following (pseudo) Pascal code which employs a loop..endloop construct:

```
        LOOP
                READ(ch)
                IF ch = '.' THEN BREAK;
                WRITE(ch);
        ENDLOOP;
```

In real Pascal, you'd use the following code to accomplish this:

```
1:
        READ(ch);
        IF ch = '.' THEN GOTO 2; (* Turbo Pascal supports BREAK! *)
        WRITE(ch);
        GOTO 1
2:
```

In assembly language you'd end up with something like:

```
LOOP1:  getc
        cmp     al, '.'
        je      EndLoop
        putc
        jmp     LOOP1
EndLoop:
```

10.6.4 FOR Loops

The for loop is a special form of the while loop which repeats the loop body a specific number of times. In Pascal, the for loop looks something like the following:

```
        FOR var := initial TO final DO stmt
or
        FOR var := initial DOWNTO final DO stmt
```

Traditionally, the for loop in Pascal has been used to process arrays and other objects accessed in sequential numeric order. These loops can be converted directly into assembly language as follows:

4. Technically, C/C++ *does* support such a loop. "for(;;)" along with break provides this capability.

5. Many high level languages use statements like NEXT, BREAK, CONTINUE, EXIT, and CYCLE rather than GOTO; but they're all forms of the GOTO statement.

In Pascal:

```
FOR var := start TO stop DO stmt;
```

In Assembly:

```
                mov     var, start
FL:             mov     ax, var
                cmp     ax, stop
                jg      EndFor

; code corresponding to stmt goes here.

                inc     var
                jmp     FL
EndFor:
```

Fortunately, most for loops repeat some statement(s) a fixed number of times. For example,

```
FOR I := 0 to 7 do write(ch);
```

In situations like this, it's better to use the 80x86 loop instruction rather than simulate a for loop:

```
                mov     cx, 7
LP:             mov     al, ch
                call    putc
                loop    LP
```

Keep in mind that the loop instruction normally appears at the end of a loop whereas the for loop tests for termination at the beginning of the loop. Therefore, you should take precautions to prevent a runaway loop in the event cx is zero (which would cause the loop instruction to repeat the loop 65,536 times) or the stop value is less than the start value. In the case of

```
FOR var := start TO stop DO stmt;
```

assuming you don't use the value of var within the loop, you'd probably want to use the assembly code:

```
                mov     cx, stop
                sub     cx, start
                jl      SkipFor
                inc     cx
LP:             stmt
                loop    LP
SkipFor:
```

Remember, the sub and cmp instructions set the flags in an identical fashion. Therefore, this loop will be skipped if stop is less than start. It will be repeated (stop-start)+1 times otherwise. If you need to reference the value of var within the loop, you could use the following code:

```
                mov     ax, start
                mov     var, ax
                mov     cx, stop
                sub     cx, ax
                jl      SkipFor
                inc     cx
LP:             stmt
                inc     var
                loop    LP
SkipFor:
```

The downto version appears in the exercises.

10.7 Register Usage and Loops

Given that the 80x86 accesses registers much faster than memory locations, registers are the ideal spot to place loop control variables (especially for small loops). This point is

amplified since the loop instruction requires the use of the cx register. However, there are some problems associated with using registers within a loop. The primary problem with using registers as loop control variables is that registers are a limited resource. In particular, there is only one cx register. Therefore, the following will not work properly:

```
                mov     cx, 8
Loop1:          mov     cx, 4
Loop2:          stmts
                loop    Loop2
                stmts
                loop    Loop1
```

The intent here, of course, was to create a set of nested loops, that is, one loop inside another. The inner loop (Loop2) should repeat four times for each of the eight executions of the outer loop (Loop1). Unfortunately, both loops use the loop instruction. Therefore, this will form an infinite loop since cx will be set to zero (which loop treats like 65,536) at the end of the first loop instruction. Since cx is always zero upon encountering the second loop instruction, control will always transfer to the Loop1 label. The solution here is to save and restore the cx register or to use a different register in place of cx for the outer loop:

```
                mov     cx, 8
Loop1:          push    cx
                mov     cx, 4
Loop2:          stmts
                loop    Loop2
                pop     cx
                stmts
                loop    Loop1
```

or:

```
                mov     bx, 8
Loop1:          mov     cx, 4
Loop2:          stmts
                loop    Loop2
                stmts
                dec     bx
                jnz     Loop1
```

Register corruption is one of the primary sources of bugs in loops in assembly language programs, always keep an eye out for this problem.

10.8 Performance Improvements

The 80x86 microprocessors execute sequences of instructions at blinding speeds. You'll rarely encounter a program that is slow which doesn't contain any loops. Since loops are the primary source of performance problems within a program, they are the place to look when attempting to speed up your software. While a treatise on how to write efficient programs is beyond the scope of this chapter, there are some things you should be aware of when designing loops in your programs. They're all aimed at removing unnecessary instructions from your loops in order to reduce the time it takes to execute one iteration of the loop.

10.8.1 Moving the Termination Condition to the End of a Loop

Consider the following flow graphs for the three types of loops presented earlier:

Repeat..until loop:

```
Initialization code
          Loop body
Test for termination
Code following the loop
```

While loop:

```
        Initialization code
        Loop termination test
                Loop body
                Jump back to test
        Code following the loop
```

Loop..endloop loop:

```
        Initialization code
                Loop body, part one
                Loop termination test
                Loop body, part two
                Jump back to loop body part 1
        Code following the loop
```

As you can see, the repeat..until loop is the simplest of the bunch. This is reflected in the assembly language code required to implement these loops. Consider the following repeat..until and while loops that are identical:

```
        SI := DI - 20;                  SI := DI - 20;
        while (SI <= DI) do             repeat
        begin

                stmts                           stmts
                SI := SI + 1;                   SI := SI + 1;

        end;                            until SI > DI;
```

The assembly language code for these two loops is[6]:

```
        mov     si, di                          mov     si, di
        sub     si, 20                          sub     si, 20
WL1:    cmp     si, di                  U:      stmts
        jnle    QWL                             inc     si
        stmts                                   cmp     si, di
        inc     si                              jng     RU
        jmp     WL1
QWL:
```

As you can see, testing for the termination condition at the end of the loop allowed us to remove a jmp instruction from the loop. This can be significant if this loop is nested inside other loops. In the preceding example there wasn't a problem with executing the body at least once. Given the definition of the loop, you can easily see that the loop will be executed exactly 20 times. Assuming cx is available, this loop easily reduces to:

```
                lea     si, -20[di]
                mov     cx, 20
WL1:            stmts
                inc     si
                loop    WL1
```

Unfortunately, it's not always quite this easy. Consider the following Pascal code:

```
        WHILE (SI <= DI) DO BEGIN
                stmts
                SI := SI + 1;
        END;
```

In this particular example, we haven't the slightest idea what si contains upon entry into the loop. Therefore, we cannot assume that the loop body will execute at least once. Therefore, we must do the test before executing the body of the loop. The test can be placed at the end of the loop with the inclusion of a single jmp instruction:

```
                jmp     short Test
RU:             stmts
                inc     si
Test:           cmp     si, di
                jle     RU
```

6. Of course, a good compiler would recognize that both loops perform the same operation and generate identical code for each. However, most compilers are not this good.

Although the code is as long as the original while loop, the jmp instruction executes only once rather than on each repetition of the loop. Note that this slight gain in efficiency is obtained via a slight loss in readability. The second code sequence above is closer to spaghetti code that the original implementation. Such is often the price of a small performance gain. Therefore, you should carefully analyze your code to ensure that the performance boost is worth the loss of clarity. More often than not, assembly language programmers sacrifice clarity for dubious gains in performance, producing impossible to understand programs.

10.8.2 Executing the Loop Backwards

Because of the nature of the flags on the 80x86, loops which range from some number down to (or up to) zero are more efficient than any other. Compare the following Pascal loops and the code they generate:

```
      for I := 1 to 8 do
                K := K + I - J;

                mov     I, 1
FLP:            mov     ax, K
                add     ax, I
                sub     ax, J
                mov     K, ax
                inc     I
                cmp     I, 8
                jle     FLP
```

```
for I := 8 downto 1 do
       K := K + I - j;

       mov    I, 8
FLP:   mov    ax, K
       add    ax, I
       sub    ax, J
       mov    K, ax
       dec    I
       jnz    FLP
```

Note that by running the loop from eight down to one (the code on the right) we saved a comparison on each repetition of the loop.

Unfortunately, you cannot force all loops to run backwards. However, with a little effort and some coercion you should be able to work most loops so they operate backwards. Once you get a loop operating backwards, it's a good candidate for the loop instruction (which will improve the performance of the loop on pre-486 CPUs).

The example above worked out well because the loop ran from eight down to one. The loop terminated when the loop control variable became zero. What happens if you need to execute the loop when the loop control variable goes to zero? For example, suppose that the loop above needed to range from seven down to zero. As long as the upper bound is positive, you can substitute the jns instruction in place of the jnz instruction above to repeat the loop some specific number of times:

```
                mov     I, 7
FLP:            mov     ax, K
                add     ax, I
                sub     ax, J
                mov     K, ax
                dec     I
                jns     FLP
```

This loop will repeat eight times with I taking on the values seven down to zero on each execution of the loop. When it decrements zero to minus one, it sets the sign flag and the loop terminates.

Keep in mind that some values may look positive but they are negative. If the loop control variable is a byte, then values in the range 128..255 are negative. Likewise, 16-bit values in the range 32768..65535 are negative. Therefore, initializing the loop control variable with any value in the range 129..255 or 32769..65535 (or, of course, zero) will cause the loop to terminate after a single execution. This can get you into a lot of trouble if you're not careful.

10.8.3 Loop Invariant Computations

A loop invariant computation is some calculation that appears within a loop that always yields the same result. You needn't do such computations inside the loop. You can compute them outside the loop and reference the value of the computation inside. The following Pascal code demonstrates a loop which contains an invariant computation:

```
FOR I := 0 TO N DO
          K := K+(I+J-2);
```

Since J never changes throughout the execution of this loop, the sub-expression "J-2" can be computed outside the loop and its value used in the expression inside the loop:

```
temp := J-2;
FOR I := 0 TO N DO
          K := K+(I+temp);
```

Of course, if you're really interested in improving the efficiency of this particular loop, you'd be much better off (most of the time) computing K using the formula:

$$K = K + ((N+1) \times temp) + \frac{(N+2) \times (N+2)}{2}$$

This computation for K is based on the formula:

$$\sum_{i=0}^{N} i = \frac{(N+1) \times (N)}{2}$$

However, simple computations such as this one aren't always possible. Still, this demonstrates that a better algorithm is almost always better than the trickiest code you can come up with.

In assembly language, invariant computations are even trickier. Consider this conversion of the Pascal code above:

```
                mov     ax, J
                add     ax, 2
                mov     temp, ax
                mov     ax, n
                mov     I, ax
FLP:            mov     ax, K
                add     ax, I
                sub     ax, temp
                mov     K, ax
                dec     I
                cmp     I, -1
                jg      FLP
```

Of course, the first refinement we can make is to move the loop control variable (I) into a register. This produces the following code:

```
                mov     ax, J
                inc     ax
                inc     ax
                mov     temp, ax
                mov     cx, n
FLP:            mov     ax, K
                add     ax, cx
                sub     ax, temp
                mov     K, ax
                dec     cx
                cmp     cx, -1
                jg      FLP
```

This operation speeds up the loop by removing a memory access from each repetition of the loop. To take this one step further, why not use a register to hold the "temp" value rather than a memory location:

```
                mov     bx, J
                inc     bx
                inc     bx
                mov     cx, n
FLP:            mov     ax, K
                add     ax, cx
                sub     ax, bx
                mov     K, ax
                dec     cx
                cmp     cx, -1
                jg      FLP
```

Furthermore, accessing the variable K can be removed from the loop as well:

```
                mov     bx, J
                inc     bx
                inc     bx
                mov     cx, n
                mov     ax, K
FLP:            add     ax, cx
                sub     ax, bx
                dec     cx
                cmp     cx, -1
                jg      FLP
                mov     K, ax
```

One final improvement which is begging to be made is to substitute the loop instruction for the dec cx / cmp cx,-1 / JG FLP instructions. Unfortunately, this loop must be repeated whenever the loop control variable hits zero, the loop instruction cannot do this. However, we can unravel the last execution of the loop (see the next section) and do that computation outside the loop as follows:

```
                mov     bx, J
                inc     bx
                inc     bx
                mov     cx, n
                mov     ax, K
FLP:            add     ax, cx
                sub     ax, bx
                loop    FLP
                sub     ax, bx
                mov     K, ax
```

As you can see, these refinements have considerably reduced the number of instructions executed inside the loop and those instructions that do appear inside the loop are very fast since they all reference registers rather than memory locations.

Removing invariant computations and unnecessary memory accesses from a loop (particularly an inner loop in a set of nested loops) can produce dramatic performance improvements in a program.

10.8.4 Unraveling Loops

For small loops, that is, those whose body is only a few statements, the overhead required to process a loop may constitute a significant percentage of the total processing time. For example, look at the following Pascal code and its associated 80x86 assembly language code:

```
        FOR I := 3 DOWNTO 0 DO A [I] := 0;

                    mov     I, 3
FLP:                mov     bx, I
                    shl     bx, 1
                    mov     A [bx], 0
                    dec     I
                    jns     FLP
```

Each execution of the loop requires five instructions. Only one instruction is performing the desired operation (moving a zero into an element of A). The remaining four instructions convert the loop control variable into an index into A and control the repetition of the loop. Therefore, it takes 20 instructions to do the operation logically required by four.

While there are many improvements we could make to this loop based on the information presented thus far, consider carefully exactly what it is that this loop is doing-- it's simply storing four zeros into A[0] through A[3]. A more efficient approach is to use four mov instructions to accomplish the same task. For example, if A is an array of words, then the following code initializes A much faster than the code above:

```
                    mov     A, 0
                    mov     A+2, 0
                    mov     A+4, 0
                    mov     A+6, 0
```

You may improve the execution speed and the size of this code by using the ax register to hold zero:

```
                    xor     ax, ax
                    mov     A, ax
                    mov     A+2, ax
                    mov     A+4, ax
                    mov     A+6, ax
```

Although this is a trivial example, it shows the benefit of loop unraveling. If this simple loop appeared buried inside a set of nested loops, the 5:1 instruction reduction could possibly double the performance of that section of your program.

Of course, you cannot unravel all loops. Loops that execute a variable number of times cannot be unraveled because there is rarely a way to determine (at assembly time) the number of times the loop will be executed. Therefore, unraveling a loop is a process best applied to loops that execute a known number of times.

Even if you repeat a loop some fixed number of iterations, it may not be a good candidate for loop unraveling. Loop unraveling produces impressive performance improvements when the number of instructions required to control the loop (and handle other overhead operations) represent a significant percentage of the total number of instructions in the loop. Had the loop above contained 36 instructions in the body of the loop (exclusive of the four overhead instructions), then the performance improvement would be, at best, only 10% (compared with the 300-400% it now enjoys). Therefore, the costs of unraveling a loop, i.e., all the extra code which must be inserted into your program, quickly reaches a point of diminishing returns as the body of the loop grows larger or as the number of iterations increases. Furthermore, entering that code into your program can become quite a chore. Therefore, loop unraveling is a technique best applied to small loops.

Note that the superscalar x86 chips (Pentium and later) have *branch prediction hardware* and use other techniques to improve performance. Loop unrolling on such systems many actually *slow down* the code since these processors are optimized to execute short loops.

10.8.5 Induction Variables

The following is a slight modification of the loop presented in the previous section:

```
        FOR I := 0 TO 255 DO A [I] := 0;

                mov     I, 0
FLP:            mov     bx, I
                shl     bx, 1
                mov     A [bx], 0
                inc     I
                cmp     I, 255
                jbe     FLP
```

Although unraveling this code will still produce a tremendous performance improvement, it will take 257 instructions to accomplish this task[7], too many for all but the most time-critical applications. However, you can reduce the execution time of the body of the loop tremendously using *induction variables*. An induction variable is one whose value depends entirely on the value of some other variable. In the example above, the index into the array A tracks the loop control variable (it's always equal to the value of the loop control variable times two). Since I doesn't appear anywhere else in the loop, there is no sense in performing all the computations on I. Why not operate directly on the array index value? The following code demonstrates this technique:

```
                mov     bx, 0
FLP:            mov     A [bx], 0
                inc     bx
                inc     bx
                cmp     bx, 510
                jbe     FLP
```

Here, several instructions accessing memory were replaced with instructions that only access registers. Another improvement to make is to shorten the MOVA[bx],0 instruction using the following code:

```
                lea     bx, A
                xor     ax, ax
FLP:            mov     [bx], ax
                inc     bx
                inc     bx
                cmp     bx, offset A+510
                jbe     FLP
```

This code transformation improves the performance of the loop even more. However, we can improve the performance even more by using the loop instruction and the cx register to eliminate the cmp instruction[8]:

```
                lea     bx, A
                xor     ax, ax
                mov     cx, 256
FLP:            mov     [bx], ax
                inc     bx
                inc     bx
                loop    FLP
```

This final transformation produces the fastest executing version of this code[9].

10.8.6 Other Performance Improvements

There are many other ways to improve the performance of a loop within your assembly language programs. For additional suggestions, a good text on compilers such as "Compilers, Principles, Techniques, and Tools" by Aho, Sethi, and Ullman would be an

7. For this particular loop, the STOSW instruction could produce a big performance improvement on many 80x86 processors. Using the STOSW instruction would require only about six instructions for this code. See the chapter on string instructions for more details.

8. The LOOP instruction is not the best choice on the 486 and Pentium processors since dec cx" followed by "jne lbl" actually executes faster.

9. Fastest is a dangerous statement to use here! But it is the fastest of the examples presented here.

excellent place to look. Additional efficiency considerations will be discussed in the volume on efficiency and optimization.

10.9 Nested Statements

As long as you stick to the templates provides in the examples presented in this chapter, it is very easy to nest statements inside one another. The secret to making sure your assembly language sequences nest well is to ensure that each construct has one entry point and one exit point. If this is the case, then you will find it easy to combine statements. All of the statements discussed in this chapter follow this rule.

Perhaps the most commonly nested statements are the if..then..else statements. To see how easy it is to nest these statements in assembly language, consider the following Pascal code:

```
if (x = y) then
          if (I >= J) then writeln('At point 1')
          else writeln('At point 2)
else write('Error condition');
```

To convert this nested if..then..else to assembly language, start with the outermost if, convert it to assembly, then work on the innermost if:

```
; if (x = y) then

                mov     ax, X
                cmp     ax, Y
                jne     Else0

; Put innermost IF here

                jmp     IfDone0

; Else write('Error condition');

Else0:          print
                byte    "Error condition",0
IfDone0:
```

As you can see, the above code handles the "if (X=Y)..." instruction, leaving a spot for the second if. Now add in the second if as follows:

```
; if (x = y) then

                mov     ax, X
                cmp     ax, Y
                jne     Else0

;       IF ( I >= J) then writeln('At point 1')

                mov     ax, I
                cmp     ax, J
                jnge    Else1
                print
                byte    "At point 1",cr,lf,0
                jmp     IfDone1

;       Else writeln ('At point 2');

Else1:          print
                byte    "At point 2",cr,lf,0
IfDone1:

                jmp     IfDone0

; Else write('Error condition');
```

```
Else0:          print
                byte    "Error condition",0
IfDone0:
```

The nested if appears in italics above just to help it stand out.

There is an obvious optimization which you do not really want to make until speed becomes a real problem. Note in the innermost if statement above that the JMP IFDONE1 instructions simply jumps to a jmp instruction which transfers control to IfDone0. It is very tempting to replace the first jmp by one which jumps directly to IFDone0. Indeed, when you go in and optimize your code, this would be a good optimization to make. However, you shouldn't make such optimizations to your code unless you really need the speed. Doing so makes your code harder to read and understand. Remember, we would like all our control structures to have one entry and one exit. Changing this jump as described would give the innermost if statement two exit points.

The for loop is another commonly nested control structure. Once again, the key to building up nested structures is to construct the outside object first and fill in the inner members afterwards. As an example, consider the following nested for loops which add the elements of a pair of two dimensional arrays together:

```
        for i := 0 to 7 do
                for k := 0 to 7 do
                        A [i,j] := B [i,j] + C [i,j];
```

As before, begin by constructing the outermost loop first. This code assumes that dx will be the loop control variable for the outermost loop (that is, dx is equivalent to "i"):

```
; for dx := 0 to 7 do

                mov     dx, 0
ForLp0:         cmp     dx, 7
                jnle    EndFor0

; Put innermost FOR loop here

                inc     dx
                jmp     ForLp0
EndFor0:
```

Now add the code for the nested for loop. Note the use of the cx register for the loop control variable on the innermost for loop of this code.

```
; for dx := 0 to 7 do

                mov     dx, 0
ForLp0:         cmp     dx, 7
                jnle    EndFor0

;       for cx := 0 to 7 do

                mov     cx, 0
ForLp1:         cmp     cx, 7
                jnle    EndFor1

; Put code for A[dx,cx] := b[dx,cx] + C [dx,cx] here

                inc     cx
                jmp     ForLp1
EndFor1:

                inc     dx
                jmp     ForLp0
EndFor0:
```

Once again the innermost for loop is in italics in the above code to make it stand out. The final step is to add the code which performs that actual computation.

10.10 Timing Delay Loops

Most of the time the computer runs too slow for most people's tastes. However, there are occasions when it actually runs *too fast*. One common solution is to create an empty loop to waste a small amount of time. In Pascal you will commonly see loops like:

```
for i := 1 to 10000 do ;
```

In assembly, you might see a comparable loop:

```
                mov     cx, 8000h
DelayLp:        loop    DelayLp
```

By carefully choosing the number of iterations, you can obtain a relatively accurate delay interval. There is, however, one catch. That relatively accurate delay interval is only going to be accurate on *your* machine. If you move your program to a different machine with a different CPU, clock speed, number of wait states, different sized cache, or half a dozen other features, you will find that your delay loop takes a completely different amount of time. Since there is better than a hundred to one difference in speed between the high end and low end PCs today, it should come as no surprise that the loop above will execute 100 times faster on some machines than on others.

The fact that one CPU runs 100 times faster than another does not reduce the need to have a delay loop which executes some fixed amount of time. Indeed, it makes the problem that much more important. Fortunately, the PC provides a hardware based timer which operates at the same speed regardless of the CPU speed. This timer maintains the time of day for the operating system, so it's very important that it run at the same speed whether you're on an 8088 or a Pentium. In the chapter on interrupts you will learn to actually patch into this device to perform various tasks. For now, we will simply take advantage of the fact that this timer chip forces the CPU to increment a 32-bit memory location (40:6ch) about 18.2 times per second. By looking at this variable we can determine the speed of the CPU and adjust the count value for an empty loop accordingly.

The basic idea of the following code is to watch the BIOS timer variable until it changes. Once it changes, start counting the number of iterations through some sort of loop until the BIOS timer variable changes again. Having noted the number of iterations, if you execute a similar loop the same number of times it should require about 1/18.2 seconds to execute.

The following program demonstrates how to create such a Delay routine:

```
                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

; PPI_B is the I/O address of the keyboard/speaker control
; port. This program accesses it simply to introduce a
; large number of wait states on faster machines. Since the
; PPI (Programmable Peripheral Interface) chip runs at about
; the same speed on all PCs, accessing this chip slows most
; machines down to within a factor of two of the slower
; machines.

PPI_B           equ     61h

; RTC is the address of the BIOS timer variable (40:6ch).
; The BIOS timer interrupt code increments this 32-bit
; location about every 55 ms (1/18.2 seconds). The code
; which initializes everything for the Delay routine
; reads this location to determine when 1/18th seconds
; have passed.

RTC             textequ  <es:[6ch]>

dseg            segment  para public 'data'
```

```
; TimedValue contains the number of iterations the delay
; loop must repeat in order to waste 1/18.2 seconds.

TimedValue      word    0

; RTC2 is a dummy variable used by the Delay routine to
; simulate accessing a BIOS variable.

RTC2            word    0


dseg            ends



cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

; Main program which tests out the DELAY subroutine.

Main            proc
                mov     ax, dseg
                mov     ds, ax

                print
                byte    "Delay test routine",cr,lf,0

; Okay, let's see how long it takes to count down 1/18th
; of a second. First, point ES as segment 40h in memory.
; The BIOS variables are all in segment 40h.
;
; This code begins by reading the memory timer variable
; and waiting until it changes. Once it changes we can
; begin timing until the next change occurs. That will
; give us 1/18.2 seconds. We cannot start timing right
; away because we might be in the middle of a 1/18.2
; second period.

                mov     ax, 40h
                mov     es, ax
                mov     ax, RTC
RTCMustChange:  cmp     ax, RTC
                je      RTCMustChange

; Okay, begin timing the number of iterations it takes
; for an 18th of a second to pass. Note that this
; code must be very similar to the code in the Delay
; routine.

                mov     cx, 0
                mov     si, RTC
                mov     dx, PPI_B
TimeRTC:        mov     bx, 10
DelayLp:        in      al, dx
                dec     bx
                jne     DelayLp
                cmp     si, RTC
                loope   TimeRTC

                neg     cx                      ;CX counted down!
                mov     TimedValue, cx          ;Save away

                mov     ax, ds
                mov     es, ax

                printf
                byte    "TimedValue = %d",cr,lf
                byte    "Press any key to continue",cr,lf
                byte    "This will begin a delay of five "
```

```
                byte    "seconds",cr,lf,0
                dword   TimedValue

                getc

                mov     cx, 90
DelayIt:        call    Delay18
                loop    DelayIt

Quit:           ExitPgm ;DOS macro to quit program.
Main            endp

; Delay18-This routine delays for approximately 1/18th sec.
;       Presumably, the variable "TimedValue" in DS has
;       been initialized with an appropriate count down
;       value before calling this code.

Delay18         proc    near
                push    ds
                push    es
                push    ax
                push    bx
                push    cx
                push    dx
                push    si

                mov     ax, dseg
                mov     es, ax
                mov     ds, ax

; The following code contains two loops. The inside
; nested loop repeats 10 times. The outside loop
; repeats the number of times determined to waste
; 1/18.2 seconds. This loop accesses the hardware
; port "PPI_B" in order to introduce many wait states
; on the faster processors. This helps even out the
; timings on very fast machines by slowing them down.
; Note that accessing PPI_B is only done to introduce
; these wait states, the data read is of no interest
; to this code.
;
; Note the similarity of this code to the code in the
; main program which initializes the TimedValue variable.

                mov     cx, TimedValue
                mov     si, es:RTC2
                mov     dx, PPI_B

TimeRTC:        mov     bx, 10
DelayLp:        in      al, dx
                dec     bx
                jne     DelayLp
                cmp     si, es:RTC2
                loope   TimeRTC

                pop     si
                pop     dx
                pop     cx
                pop     bx
                pop     ax
                pop     es
                pop     ds
                ret
Delay18         endp

cseg            ends

sseg            segment para stack 'stack'
stk             word    1024 dup (0)
sseg            ends
                end     Main
```

10.11 Sample Program

This chapter's sample program is a simple moon lander game. While the simulation isn't terribly realistic, this program does demonstrate the use and optimization of several different control structures including loops, if..then..else statements, and so on.

```
; Simple "Moon Lander" game.
;
; Randall Hyde
; 2/8/96
;
; This program is an example of a trivial little "moon lander"
; game that simulates a Lunar Module setting down on the Moon's
; surface.  At time T=0 the spacecraft's velocity is 1000 ft/sec
; downward, the craft has 1000 units of fuel, and the craft is
; 10,000 ft above the moon's surface.  The pilot (user) can
; specify how much fuel to burn at each second.
;
; Note that all calculations are approximate since everything is
; done with integer arithmetic.


; Some important constants

InitialVelocity  =        1000
InitialDistance  =        10000
InitialFuel      =        250
MaxFuelBurn      =        25

MoonsGravity     =        5            ;Approx 5 ft/sec/sec
AccPerUnitFuel   =        -5           ;-5 ft/sec/sec for each fuel unit.


                 .xlist
                 include    stdlib.a
                 includelib stdlib.lib
                 .list

dseg             segment  para public 'data'

; Current distance from the Moon's Surface:

CurDist          word     InitialDistance

; Current Velocity:

CurVel           word     InitialVelocity

; Total fuel left to burn:

FuelLeft         word     InitialFuel


; Amount of Fuel to use on current burn.

Fuel             word     ?

; Distance travelled in the last second.

Dist             word     ?

dseg             ends



cseg             segment  para public 'code'
                 assume   cs:cseg, ds:dseg
```

```
; GETI-Reads an integer variable from the user and returns its
;      its value in the AX register.  If the user entered garbage,
;      this code will make the user re-enter the value.

geti            textequ  <call _geti>
_geti           proc
                push     es
                push     di
                push     bx

; Read a string of characters from the user.
;
; Note that there are two (nested) loops here.  The outer loop
; (GetILp) repeats the getsm operation as long as the user
; keeps entering an invalid number.  The innermost loop (ChkDigits)
; checks the individual characters in the input string to make
; sure they are all decimal digits.

GetILp:         getsm

; Check to see if this string contains any non-digit characters:
;
; while (([bx] >= '0') and ([bx] <= '9')  bx := bx + 1;
;
; Note the sneaky way of turning the while loop into a
; repeat..until loop.

                mov      bx, di       ;Pointer to start of string.
                dec      bx
ChkDigits:      inc      bx
                mov      al, es:[bx]  ;Fetch next character.
                IsDigit               ;See if it's a decimal digit.
                je       ChkDigits    ;Repeat if it is.

                cmp      al, 0        ;At end of string?
                je       GotNumber

; Okay, we just ran into a non-digit character.  Complain and make
; the user reenter the value.

                free                  ;Free space malloc'd by getsm.
                print
                byte     cr,lf
                byte     "Illegal unsigned integer value, "
                byte     "please reenter.",cr,lf
                byte     "(no spaces, non-digit chars, etc.):",0
                jmp      GetILp


; Okay, ES:DI is pointing at something resembling a number.  Convert
; it to an integer.

GotNumber:      atoi
                free                  ;Free space malloc'd by getsm.

                pop      bx
                pop      di
                pop      es
                ret
_geti           endp


; InitGame-      Initializes global variables this game uses.

InitGame        proc
                mov      CurVel, InitialVelocity
                mov      CurDist, InitialDistance
                mov      FuelLeft, InitialFuel
                mov      Dist, 0
                ret
```

```
InitGame        endp


; DispStatus-    Displays important information for each
;                cycle of the game (a cycle is one second).

DispStatus      proc
                printf
                byte    cr,lf
                byte    "Distance from surface: %5d",cr,lf
                byte    "Current velocity:      %5d",cr,lf
                byte    "Fuel left:             %5d",cr,lf
                byte    lf
                byte    "Dist travelled in the last second: %d",cr,lf
                byte    lf,0
                dword   CurDist, CurVel, FuelLeft, Dist
                ret
DispStatus      endp


; GetFuel-       Reads an integer value representing the amount of fuel
;                to burn from the user and checks to see if this value
;                is reasonable.  A reasonable value must:
;
;                * Be an actual number (GETI handles this).
;                * Be greater than or equal to zero (no burning
;                  negative amounts of fuel, GETI handles this).
;                * Be less than MaxFuelBurn (any more than this and
;                  you have an explosion, not a burn).
;                * Be less than the fuel left in the Lunar Module.

GetFuel         proc
                push    ax

; Loop..endloop structure that reads an integer input and terminates
; if the input is reasonable.  It prints a message an repeats if
; the input is not reasonable.
;
; loop
;       get fuel;
;       if (fuel < MaxFuelBurn) then break;
;       print error message.
; endloop
;
; if (fuel > FuelLeft) then
;
;       fuel = fuelleft;
;       print appropriate message.
;
; endif

GetFuelLp:      print
                byte    "Enter amount of fuel to burn: ",0
                geti
                cmp     ax, MaxFuelBurn
                jbe     GoodFuel

                print
                byte    "The amount you've specified exceeds the "
                byte    "engine rating,", cr, lf
                byte    "please enter a smaller value",cr,lf,lf,0
                jmp     GetFuelLp

GoodFuel:       mov     Fuel, ax
                cmp     ax, FuelLeft
                jbe     HasEnough
                printf
                byte    "There are only %d units of fuel left.",cr,lf
                byte    "The Lunar module will burn this rather than %d"
                byte    cr,lf,0
                dword   FuelLeft, Fuel

                mov     ax, FuelLeft
```

```
                mov     Fuel, ax

HasEnough:      mov     ax, FuelLeft
                sub     ax, Fuel
                mov     FuelLeft, ax
                pop     ax
                ret
GetFuel         endp

; ComputeStatus-
;
;       This routine computes the new velocity and new distance based on the
;       current distance, current velocity, fuel burnt, and the moon's
;       gravity.  This routine is called for every "second" of flight time.
;       This simplifies the following equations since the value of T is
;       always one.
;
; note:
;
;       Distance Travelled = Acc*T*T/2 + Vel*T  (note: T=1, so it goes away).
;       Acc = MoonsGravity + Fuel * AccPerUnitFuel
;
;       New Velocity = Acc*T + Prev Velocity
;
;       This code should really average these values over the one second
;       time period, but the simulation is so crude anyway, there's no
;       need to really bother.

ComputeStatus   proc
                push    ax
                push    bx
                push    dx

; First, compute the acceleration value based on the fuel burnt
; during this second (Acc = Moon's Gravity + Fuel * AccPerUnitFuel).

                mov     ax, Fuel                ;Compute
                mov     dx, AccPerUnitFuel      ; Fuel*AccPerUnitFuel
                imul    dx

                add     ax, MoonsGravity        ;Add in Moon's gravity.
                mov     bx, ax                  ;Save Acc value.

; Now compute the new velocity (V=AT+V)

                add     ax, CurVel              ;Compute new velocity
                mov     CurVel, ax

; Next, compute the distance travelled (D = 1/2 * A * T^2 + VT +D)

                sar     bx, 1                   ;Acc/2
                add     ax, bx                  ;Acc/2 + V (T=1!)
                mov     Dist, ax                ;Distance Travelled.
                neg     ax
                add     CurDist, ax             ;New distance.

                pop     dx
                pop     bx
                pop     ax
                ret
ComputeStatus   endp

; GetYorN-      Reads a yes or no answer from the user (Y, y, N, or n).
;               Returns the character read in the al register (Y or N,
;               converted to upper case if necessary).

GetYorN         proc
                getc
                ToUpper
                cmp     al, 'Y'
                je      GotIt
                cmp     al, 'N'
                jne     GetYorN
GotIt:          ret
```

```
GetYorN         endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

MoonLoop:       print
                byte    cr,lf,lf
                byte    "Welcome to the moon lander game.",cr,lf,lf
                byte    "You must manuever your craft so that you touch"
                byte    "down at less than 10 ft/sec",cr,lf
                byte    "for a soft landing.",cr,lf,lf,0

                call    InitGame

; The following loop repeats while the distance to the surface is greater
; than zero.

WhileStillUp:   mov     ax, CurDist
                cmp     ax, 0
                jle     Landed

                call    DispStatus
                call    GetFuel
                call    ComputeStatus
                jmp     WhileStillUp

Landed:         cmp     CurVel, 10
                jle     SoftLanding

                printf
                byte    "Your current velocity is %d.",cr,lf
                byte    "That was just a little too fast.  However, as a "
                byte    "consolation prize,",cr,lf
                byte    "we will name the new crater you just created "
                byte    "after you.",cr,lf,0
                dword   CurVel

                jmp     TryAgain

SoftLanding:    printf
                byte    "Congrats!  You landed the Lunar Module safely at "
                byte    "%d ft/sec.",cr,lf
                byte    "You have %d units of fuel left.",cr,lf
                byte    "Good job!",cr,lf,0
                dword   CurVel, FuelLeft

TryAgain:       print
                byte    "Do you want to try again (Y/N)? ",0
                call    GetYorN
                cmp     al, 'Y'
                je      MoonLoop

                print
                byte    cr,lf
                byte    "Thanks for playing!  Come back to the moon “
                byte    “again sometime"
                byte    cr,lf,lf,0

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

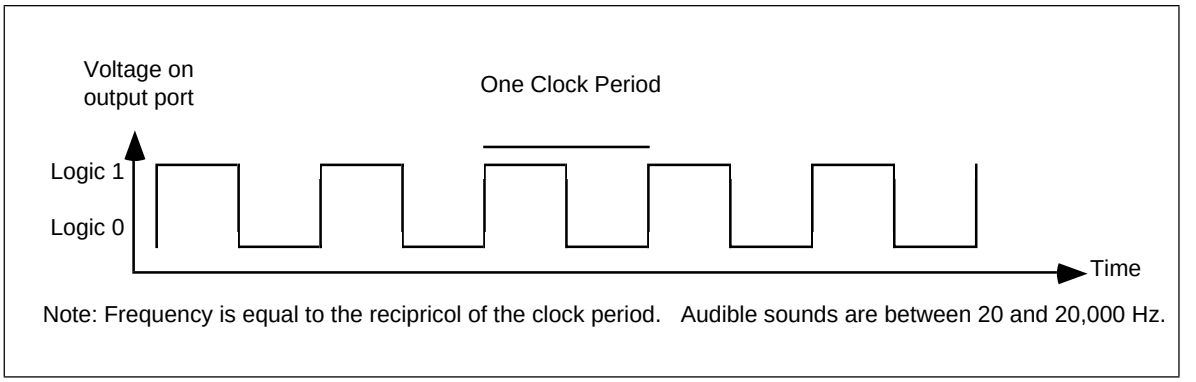


Figure 10.2 An Audible Sound Wave: The Relationship Between Period and Frequency

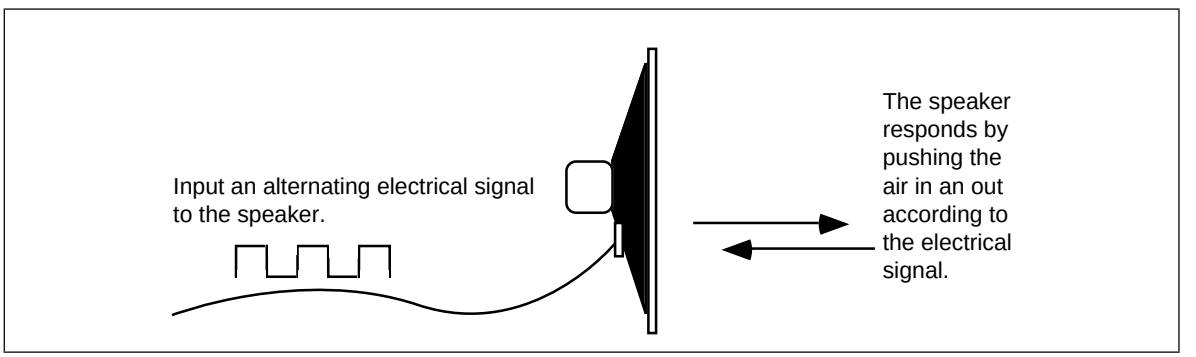


Figure 10.3 A Speaker

10.12 Laboratory Exercises

In this laboratory exercise you will program the timer chip on the PC to produce musical tones. You will learn how the PC generates sound and how you can use this ability to encode and play music.

10.12.1The Physics of Sound

Sounds you hear are the result of vibrating air molecules. When air molecules quickly vibrate back and forth between 20 and 20,000 times per second, we interpret this as some sort of sound. A *speaker* (see Figure 10.3) is a device which vibrates air in response to an electrical signal. That is, it converts an electric signal which alternates between 20 and 20,000 times per second (Hz) to an audible tone. Alternating a signal is very easy on a computer, all you have to do is apply a logic one to an output port for some period of time and then write a logic zero to the output port for a short period. Then repeat this over and over again. A plot of this activity over time appears in Figure 10.2.

Although many humans are capable of hearing tones in the range 20-20Khz, the PC's speaker is not capable of faithfully reproducing the tones in this range. It works pretty good for sounds in the range 100-10Khz, but the volume drops off dramatically outside this range. Fortunately, this lab only requires frequencies in the 110-2,000 hz range; well within the capabilities of the PC speaker.

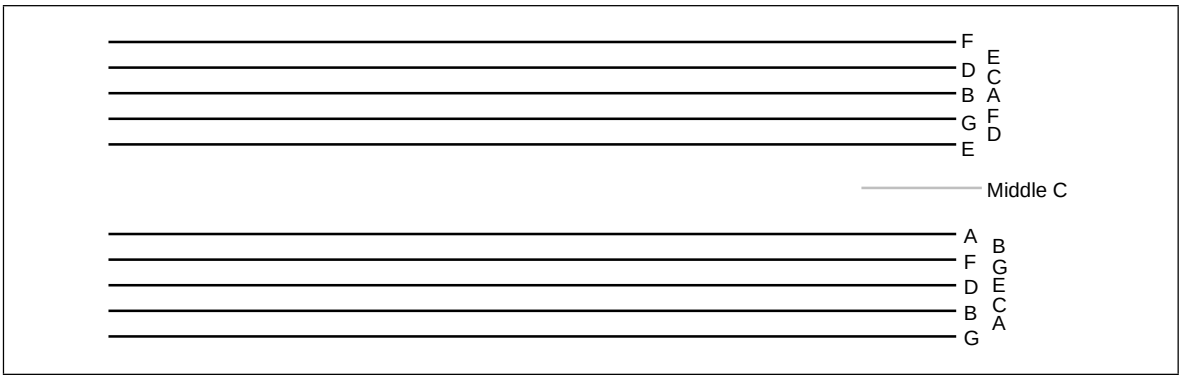


Figure 10.4 A Musical Staff

10.12.2 The Fundamentals of Music

In this laboratory you will use the timer chip and the PC's built-in speaker to produce musical tones. To produce true music, rather than annoying tones, requires a little knowledge of music theory. This section provides a very brief introduction to the notation musicians use. This will help you when you attempt to convert music in standard notation to a form the computer can use.

Western music tends to use notation based on the alphabetic letters A…G. There are a total of 12 notes designated A, A#, B, C, C#, D, D#, E, F, F#, G, and G# [10]. On a typical musical instrument these 12 notes repeat over and over again. For example, a typical piano might have six repetitions of these 12 notes. Each repetition is an *octave*. An octave is just a collection of 12 notes, it need not necessarily start with A, indeed, most pianos start with C. Although there are, technically, about 12 octaves within the normal hearing range of adults, very little music uses more than four or five octaves. In the laboratory, you will implement four octaves.

Written music typically uses two *staffs*. A staff is a set of five parallel lines. The upper staff is often called the *treble* staff and the lower staff is often called the *bass* staff. An examples appears in Figure 10.4.

A musical note, as the notation to the side of the staffs above indicates, appears both on the lines of the staffs and the spaces between the staffs. The position of the notes on the staffs determine which note to play, the *shape* of the note determines its duration. There are *whole* notes, *half* notes, *quarter* notes, *eighth* notes, *sixteenth* notes, and *thirty-second* notes[11]. Note durations are specified relative to one another. So a half note plays for one-half the time of a whole note, a quarter note plays for one-half the time of a half note (one quarter the time of a whole note), etc. In most musical passages, the quarter note is generally the basis for timing. If the *tempo* of a particular piece is 100 *beats per second* this means that you play 100 quarter notes per second.

The duration of a note is determined by its shape as shown in Figure 10.5.

In addition to the notes themselves, there are often brief pauses in a musical passage when there are no notes being played. These pauses are known as rests. Since there is nothing audible about them, only their duration matters. The duration of the various rests is the same as the normal notes; there are whole rests, half rests, quarter rests, etc. The symbols for these rests appear in .

This is but a brief introduction to music notation. Barely sufficient for those without any music training to convert a piece of sheet music into a form suitable for a computer

10. The notes with the "#" (pronounced *sharp*) correspond to the black keys on the piano. The other notes correspond to the white keys on the piano. Note that western music notation also describes *flats* in addition to sharps. A# is equal to B*b* (*b* denotes flat), C# corresponds to D*b*, etc. Technically, B is equivalent to C*b* and C is equivalent to B# but you will rarely see musicians refer to these notes this way.

11. The only reason their aren't shorter notes is because it would be hard to play one note which is 1/64th the length of another.

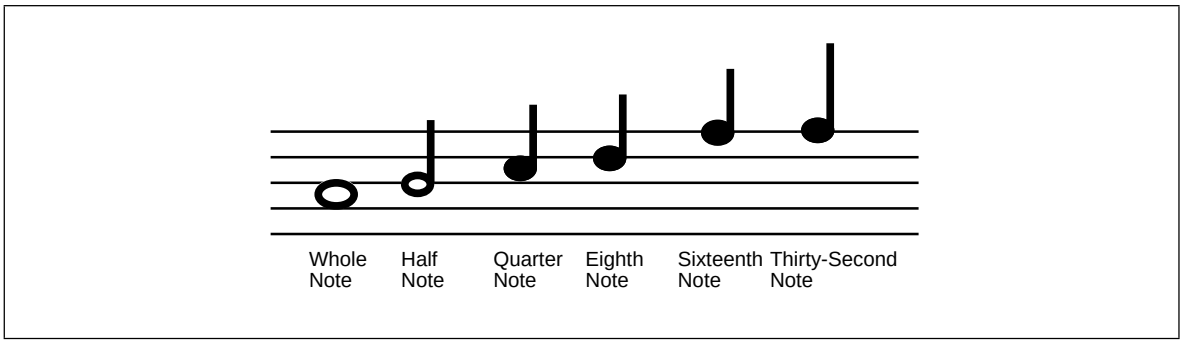


Figure 10.5 Note Durations

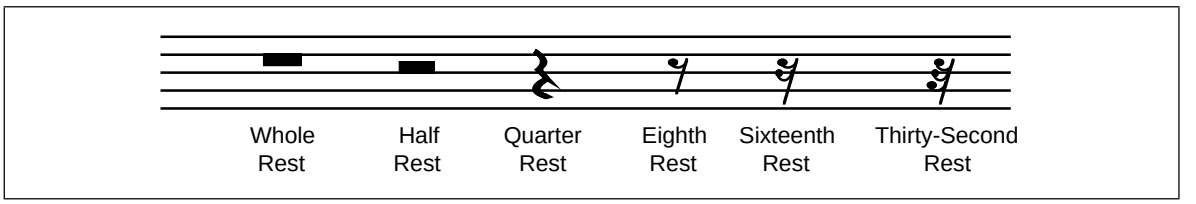


Figure 10.6 Rest Durations

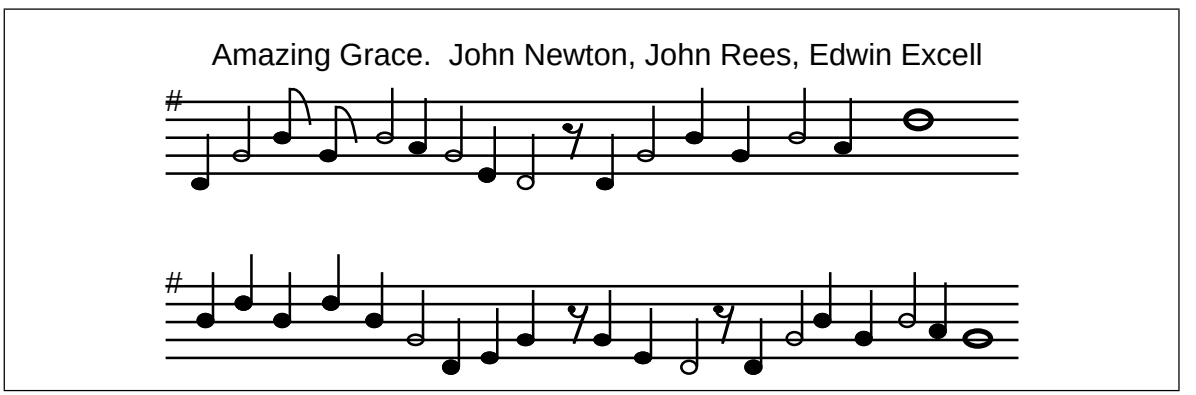


Figure 10.7 Amazing Grace

program. If you are interested in more information on music notation, the library is a good source of information on music theory.

Figure 10.7 provides an adaptation of the hymn “Amazing Grace”. There are two things to note here. First, there is no bass staff, just two treble staffs. Second, the sharp symbol on the “F” line indicates that this song is played in “G-Major” and that all F notes should be F#. There are no F notes in this song, so that hardly matters[12].

10.12.3The Physics of Music

Each musical note corresponds to a unique frequency. The A above middle C is generally 440 Hz (this is known as concert pitch since this is the frequency orchestras tune to). The A one octave below this is at 220 Hz, the A above this is 880Hz. In general, to get the next higher A you double the current frequency, to get the previous A you halve the current frequency. To obtain the remaining notes you multiply the frequency of A with a multiple of the twelfth root of two. For example, to get A# you would take the frequency for A

12. In the full version of the song there are F notes on the base clef.

and multiply it by the twelfth root of two. Repeating this operation yields the following (truncated) frequencies for four separate octaves:

Note	Frequency	Note	Frequency	Note	Frequency	Note	Frequency
A 0	110	A 1	220	A 2	440	A 3	880
A # 0	117	A # 1	233	A # 2	466	A # 3	932
B 0	123	B 1	247	B 2	494	B 3	988
C 0	131	C 1	262	C 2	523	C 3	1047
C # 0	139	C # 1	277	C # 2	554	C # 3	1109
D 0	147	D 1	294	D 2	587	D 3	1175
D # 0	156	D # 1	311	D # 2	622	D # 3	1245
E 0	165	E 1	330	E 2	659	E 3	1319
F 0	175	F 1	349	F 2	698	F 3	1397
F # 0	185	F # 1	370	F # 2	740	F # 3	1480
G 0	196	G 1	392	G 2	784	G 3	1568
G # 0	208	G # 1	415	G # 2	831	G # 3	1661

Notes: The number following each note denotes its octave. In the chart above, middle C is C1.

You can generate additional notes by halving or doubling the notes above. For example, if you really need A(-1) (the octave below A0 above), dividing the frequency of A0 by two yields 55Hz. Likewise, if you want E4, you can obtain this by doubling E3 to produce 2638 Hz. Keep in mind that the frequencies above are not exact. They are rounded to the nearest integer because we will need integer frequencies in this lab.

10.12.4 The 8253/8254 Timer Chip

PCs contain a special integrated circuit which produces a period signal. This chip (an Intel compatible 8253 or 8254, depending on your particular computer[13]) contains three different 16-bit counter/timer circuits. The PC uses one of these timers to generate the 1/18.2 second real time clock mentioned earlier. It uses the second of these timers to control the DMA refresh on main memory[14]. The third timer circuit on this chip is connected to the PC's speaker. The PC uses this timer to produces beeps, tones, and other sounds. The RTC timer will be of interest to us in a later chapter. The DMA timer, if present on your PC, isn't something you should mess with. The third timer, connected to the speaker, is the subject of this section.

10.12.5Programming the Timer Chip to Produce Musical Tones

As mentioned earlier, one of the channels on the PC programmable interval timer (PIT) chip is connected to the PC's speaker. To produce a musical tone we need to program this timer chip to produce the frequency of some desired note and then activate the

13. Most modern computers don't actually have an 8253 or 8254 chip. Instead, there is a compatible device built into some other VLSI chip on the motherboard.

14. Many modern computer systems do not use this timer for this purpose and, therefore, do not include the second timer in their chipset.

speaker. Once you initialize the timer and speaker in this fashion, the PC will continuously produce the specified tone until you disable the speaker.

To activate the speaker you must set bits zero and one of the "B Port" on the PC's 8255 Programmable Peripheral Interface (PPI) chip. Port B of the PPI is an eight-bit I/O device located at I/O address 61h. You must use the in instruction to read this port and the out instruction to write data back to it. You must preserve all other bits at this I/O address. If you modify any of the other bits, you will probably cause the PC to malfunction, perhaps even reset. The following code shows how to set bits zero and one without affecting the other bits on the port:

```
in      al, PPI_B    ;PPI_B is equated to 61h
or      al, 3        ;Set bits zero and one.
out     PPI_B, al
```

Since PPI_B's port address is less than 100h we can access this port directly, we do not have to load its port address into dx and access the port indirectly through dx.

To deactivate the speaker you must write zeros to bits zero and one of PPI_B. The code is similar to the above except you force the bits to zero rather than to one.

Manipulating bits zero and one of the PPI_B port let you turn on and off the speaker. It does not let you adjust the frequency of the tone the speaker produces. To do this you must program the PIT at I/O addresses 42h and 43h. To change the frequency applied to the speaker you must first write the value 0B6h to I/O port 43h (the PIT *control word*) and then you must write a 16-bit frequency divisor to port 42h (timer channel two). Since the port is only an eight-bit port, you must write the data using two successive OUT instructions to the same I/O address. The first byte you write is the L.O. byte of the divisor, the second byte you write is the H.O. byte.

To compute the divisor value, you must use the following formula:

$$\frac{1193180}{Frequency} = Divisor$$

For example, the divisor for the A above middle C (440 Hz) is 1,193,180/440 or 2,712 (rounded to the nearest integer). To program the PIT to play this note you would execute the following code:

```
mov     al, 0B6h     ;Control word code.
out     PIT_CW, al   ;Write control word (port 43h).
mov     al, 98h      ;2712 is 0A98h.
out     PIT_Ch2, al  ;Write L.O. byte (port 42h).
mov     al, 0ah
out     PIT_Ch2, al  ;Write H.O. byte (port 42h).
```

Assuming that you have activated the speaker, the code above will produce the A note until you deactivate the speaker or reprogram the PIT with a different divisor.

10.12.6Putting it All Together

To create *music* you will need to activate the speaker, program the PIT, and then delay for some period of time while the note plays. At the end of that period, you need to reprogram the PIT and wait while the next note plays. If you encounter a rest, you need to deactivate the speaker for the given time interval. The key point is this *time interval*. If you simply reprogram the PPI and PIT chips at microprocessor speeds, your song will be over and done with in just a few microseconds. Far to fast to hear anything. Therefore, we need to use a delay, such as the software delay code presented earlier, to allow us to hear our notes.

A reasonable tempo is between 80 and 120 quarter notes per second. This means you should be calling the Delay18 routine between 9 and 14 times for each quarter note. A reasonable set of iterations is

- three times for sixteenth notes,
- six times for eighth notes,
- twelve times for quarter notes,
- twenty-four times for half notes, and
- forty-eight times for whole notes.

Of course, you may adjust these timings as you see fit to make your music sound better. The important parameter is the ratio between the different notes and rests, not the actual time.

Since a typical piece of music contains many, many individual notes, it doesn't make sense to reprogram the PIT and PPI chips individually for each note. Instead, you should write a procedure into which you pass a divisor and a count down value. That procedure would then play that note for the specified time and then return. Assuming you call this procedure *PlayNote* and it expects the divisor in ax and the duration (number of times to call Delay18) in cx , you could use the following macro to easily create songs in your programs:

```
Note            macro    divisor, duration
                mov      ax, divisor
                mov      cx, duration
                call     PlayNote
                endm
```

The following macro lets you easily insert a rest into your music:

```
Rest            macro    Duration
                local    LoopLbl
                mov      cx, Duration
LoopLbl:        call     Delay18
                loop     LoopLbl
                endm
```

Now you can play notes by simply stringing together a list of these macros with the appropriate parameters.

The only problem with this approach is that it is different to create songs if you must constantly supply divisor values. You'll find music creation to be much simpler if you could specify the note, octave, and duration rather than a divisor and duration. This is very easy to do. Simply create a *lookup table* using the following definition:

```
Divisors: array [Note, Sharp, Octave] of word;
```

Where Note is 'A';.."G", Sharp is true or false (1 or 0), and Octave is 0..3. Each entry in the table would contain the divisor for that particular note.

10.12.7 Amazing Grace Exercise

Program Ex10_1.asm on the companion CD-ROM is a complete working program that plays the tune "Amazing Grace." Load this program an execute it.

For your lab report: the Ex10_1.asm file uses a "Note" macro that is very similar to the one appearing in the previous section. What is the difference between Ex10_1's Note macro and the one in the previous section? What changes were made to PlayNote in order to accommodate this difference?

The Ex10_1.asm program uses *straight-line code* (no loops or decisions) to play its tune. Rewrite the main body of the loop to use a pair of tables to feed the data to the Note and Rest macros. One table should contain a list of frequency values (use -1 for a rest), the other table should contain duration values. Put the two tables in the data segment and ini-

tialize them with the values for the Amazing Grace song. The loop should fetch a pair of values, one from each of the tables and call the Note or Rest macro as appropriate. When the loop encounters a frequency value of zero it should terminate. **Note:** you must call the rest macro at the end of the tune in order to shut the speaker off.

For your lab report: make the changes to the program, document them, and include the print-out of the new program in your lab report.

10.13 Programming Projects

1) Write a program to transpose two 4x4 arrays. The algorithm to transpose the arrays is

```
for i := 0 to 3 do
          for j := 0 to 3 do begin

                    temp := A [i,j];
                    A [i,j] := B [j,i];
                    B [j,i] := temp;
          end;
```

Write a main program that calls a transpose procedure. The main program should read the A array values from the user and print the A and B arrays after computing the transpose of A and placing the result in B.

2) Create a program to play music which is supplied as a string to the program. The notes to play should consist of a string of ASCII characters terminated with a byte containing the value zero. Each note should take the following form:

```
(Note)(Octave)(Duration)
```

where “Note” is A..G (upper or lower case), “Octave” is 0..3, and “Duration” is 1..8. “1” corresponds to an eighth note, “2” corresponds to a quarter note, “4” corresponds to a half note, and “8” corresponds to a whole note.

Rests consist of an explanation point followed by a “Duration” value.

Your program should ignore any spaces appearing in the string.

The following sample piece is the song “Amazing Grace” presented earlier.

```
Music           byte    "d12 g14 b11 g11 b14 a12 g14 e12 d13 !1 d12 "
                byte    "g14 b11 g11 b14 a12 d28"
                byte    "b12 d23 b11 d21 b11 g14 d12 e13 g12 e11 "
                byte    "d13 !1 d12 g14 b11 g11 b14 a12 g18"
                byte    0
```

Write a program to play any song appearing in string form like the above string. Using music obtained from another source, submit your program playing that other song.

3) A *C character string* is a sequence of characters that end with a byte containing zero. Some common character string routines include computing the length of a character string (by counting all the characters in a string up to, but not including, the zero byte), comparing two strings for equality (by comparing corresponding characters in two strings, character by character until you encounter a zero byte or two characters that are not the same), and copying one string to another (by copying the characters from one string to the corresponding positions in the other until you encounter the zero byte). Write a program that reads two strings from the user, computes the length of the first of these, compares the two strings, and then copies the first string over the top of the second. Allow for a maximum of 128 characters (including the zero byte) in your strings. Note: do not use the Standard Library string routines for this project.

4) Modify the moon lander game appearing in the Sample Programs section of this chapter (moon.asm on the companion CD-ROM, also see “Sample Program” on page 547) to allow the user to specify the initial velocity, starting distance from the surface, and initial fuel values. Verify that the values are reasonable before allowing the game to proceed.

10.14 Summary

This chapter discussed the implementation of different control structures in an assembly language programs including conditional statements (if..then..else and case statements), state machines, and iterations (loops, including while, repeat..until (do/while), loop..endloop, and for). While assembly language gives you the flexibility to create totally custom control structures, doing so often produces programs that are difficult to read and understand. Unless the situation absolutely requires something different, you should attempt to model your assembly language control structures after those in high level languages as much as possible.

The most common control structure found in high level language programs is the IF..THEN..ELSE statement. You can easily synthesize(if..then and (if..then..else statements in assembly language using the cmp instruction, the conditional jumps, and the jmp instruction. To see how to convert HLL if..then..else statements into assembly language, check out

- "IF..THEN..ELSE Sequences" on page 522

A second popular HLL conditional statement is the case (switch) statement. The case statement provides an efficient way to transfer control to one of many different statements depending on the value of some expression. While there are many ways to implement the case statement in assembly language, the most common way is to use a *jump table*. For case statements with contiguous values, this is probably the best implementation. For case statements that have widely spaced, non-contiguous values, an if..then..else implementation or some other technique is probably best. For details, see

- "CASE Statements" on page 525

State machines provide a useful paradigm for certain programming situations. A section of code which implements a state machine maintains a history of prior execution within a state variable. Subsequent execution of the code picks up in a possibly different "state" depending on prior execution. Indirect jumps provide an efficient mechanism for implementing state machines in assembly language. This chapter provided a brief introduction to state machines. To see how to implement a state machine with an indirect jump, see

- "State Machines and Indirect Jumps" on page 529

Assembly language provides some very powerful primitives for constructing a wide variety of control structures. Although this chapter concentrates on simulating HLL constructs, you can build any convoluted control structure you care to from the 80x86's cmp instruction and conditional branches. Unfortunately, the result may be very difficult to understand, especially by someone other than the original author. Although assembly language gives you the freedom to do anything you want, a mature programmer exercises restraint and chooses only those control flows which are easy to read and understand; never settling for convoluted code unless absolutely necessary. For a further description and additional guidelines, see

- "Spaghetti Code" on page 531

Iteration is one of the three basic components to programming language built around Von Neumann machines[15]. Loop control structures provide the basic iteration mechanism in most HLLs. Assembly language does not provide any looping primitives. Even the 80x86 loop instruction isn't really a loop, it's just a decrement, compare, and branch instruction. Nonetheless, it is very easy to synthesize common loop control structures in assembly language. The following sections describe how to construct HLL loop control structures in assembly language:

- "Loops" on page 531
- "While Loops" on page 532
- "Repeat..Until Loops" on page 532

15. The other two being conditional execution and the sequence.

- “LOOP..ENDLOOP Loops” on page 533
- “FOR Loops” on page 533

Program loops often consume most of the CPU time in a typical program. Therefore, if you want to improve the performance of your programs, the loops are the first place you want to look. This chapter provides several suggestions to help improve the performance of certain types of loops in assembly language programs. While they do not provide a complete guide to optimization, the following sections provide common techniques used by compilers and experienced assembly language programmers:

- “Register Usage and Loops” on page 534
- “Performance Improvements” on page 535
- “Moving the Termination Condition to the End of a Loop” on page 535
- “Executing the Loop Backwards” on page 537
- “Loop Invariant Computations” on page 538
- “Unraveling Loops” on page 539
- “Induction Variables” on page 540
- “Other Performance Improvements” on page 541

10.15 Questions

1) Convert the following Pascal statements to assembly language: (assume all variables are two byte signed integers)

   a) IF (X=Y) then A := B;

   b) IF (X <= Y) then X := X + 1 ELSE Y := Y - 1;

   c) IF NOT ((X=Y) and (Z <> T)) then Z := T else X := T;

   d) IF (X=0) and ((Y-2) > 1) then Y := Y - 1;

2) Convert the following CASE statement to assembly language:

```
CASE I OF
        0: I := 5;
        1: J := J+1;
        2: K := I+J;
        3: K := I-J;
        Otherwise I := 0;
END;
```

3) Which implementation method for the CASE statement (jump table or IF form) produces the least amount of code (including the jump table, if used) for the following CASE statements?

a)

```
CASE I OF
        0:stmt;
        100:stmt;
        1000:stmt;
END;
```

b)

```
CASE I OF
        0:stmt;
        1:stmt;
        2:stmt;
        3:stmt;
        4:stmt;
END;
```

4) For question three, which form produces the fastest code?

5) Implement the CASE statements in problem three using 80x86 assembly language.

6) What three components compose a loop?

7) What is the major difference between the WHILE, REPEAT..UNTIL, and LOOP..ENDLOOP loops?

8) What is a loop control variable?

9) Convert the following WHILE loops to assembly language: (Note: don't optimize these loops, stick exactly to the WHILE loop format)

   a)
```
I := 0;
WHILE (I < 100) DO I := I + 1;
```

   b)
```
CH := ' ';
WHILE (CH <> '.') DO BEGIN
        CH := GETC;
        PUTC(CH);
END;
```

10) Convert the following REPEAT..UNTIL loops into assembly language: (Stick exactly to the REPEAT..UNTIL loop format)

a)
```
I := 0;
REPEAT
        I := I + 1;
UNTIL   I >= 100;
```

b)
```
REPEAT
        CH := GETC;
        PUTC(CH);
UNTIL CH = '.';
```

11) Convert the following LOOP..ENDLOOP loops into assembly language: (Stick exactly to the LOOP..ENDLOOP format)

a)
```
I := 0;  LOOP
        I := I + 1;        IF I >= 100 THEN BREAK;
ENDLOOP;
```

b)
```
LOOP
        CH := GETC;    IF CH = '.' THEN BREAK; PUTC(CH);
ENDLOOP;
```

12) What are the differences, if any, between the loops in problems 4, 5, and 6? Do they perform the same operations? Which versions are most efficient?

13) Rewrite the two loops presented in the previous examples, in assembly language, as efficiently as you can.

14) By simply adding a JMP instruction, convert the two loops in problem four into REPEAT..UNTIL loops.

15) By simply adding a JMP instruction, convert the two loops in problem five to WHILE loops.

16) Convert the following FOR loops into assembly language (Note: feel free to use any of the routines provided in the UCR Standard Library package):

a) FOR I := 0 to 100 do WriteLn(I);

b)
```
FOR I := 0 to 7 do
        FOR J := 0 to 7 do
                K := K*(I-J);
```

c)
```
FOR I := 255 to 16 do
        A [I] := A[240-I]-I;
```

17) The DOWNTO reserved word, when used in conjunction with the Pascal FOR loop, runs a loop counter from a higher number down to a lower number. A FOR loop with the DOWNTO reserved word is equivalent to the following WHILE loop:

```
loopvar := initial;
while (loopvar >= final) do begin
        stmt;
        loopvar := loopvar-1;
end;
```

Implement the following Pascal FOR loops in assembly language:

a) FOR I := start downto stop do WriteLn(I);

b)
```
FOR I := 7 downto 0 do
        FOR J := 0 to 7 do
```

```
            K := K*(I-J);
```

c) FOR I := 255 downto 16 do

```
            A [I] := A[240-I]-I;
```

18) Rewrite the loop in problem 11b maintaining I in BX, J in CX, and K in AX.

19) How does moving the loop termination test to the end of the loop improve the performance of that loop?

20) What is a loop invariant computation?

21) How does executing a loop backwards improve the performance of the loop?

22) What does unraveling a loop mean?

23) How does unraveling a loop improve the loop's performance?

24) Give an example of a loop that cannot be unraveled.

25) Give an example of a loop that can be but shouldn't be unraveled.

Procedures and Functions Chapter 11

Modular design is one of the cornerstones of structured programming. A modular program contains blocks of code with single entry and exit points. You can *reuse* well written sections of code in other programs or in other sections of an existing program. If you reuse an existing segment of code, you needn't design, code, nor debug that section of code since (presumably) you've already done so. Given the rising costs of software development, modular design will become more important as time passes.

The basic unit of a modular program is the module. Modules have different meanings to different people, herein you can assume that the terms module, subprogram, subroutine, program unit, procedure, and function are all synonymous.

The procedure is the basis for a programming style. The procedural languages include Pascal, BASIC, C++, FORTRAN, PL/I, and ALGOL. Examples of non-procedural languages include APL, LISP, SNOBOL4 ICON, FORTH, SETL, PROLOG, and others that are based on other programming constructs such as functional abstraction or pattern matching. Assembly language is capable of acting as a procedural or non-procedural language. Since you're probably much more familiar with the procedural programming paradigm this text will stick to simulating procedural constructs in 80x86 assembly language.

11.0 Chapter Overview

This chapter presents an introduction to procedures and functions in assembly language. It discusses basic principles, parameter passing, function results, local variables, and recursion. You will use most of the techniques this chapter discusses in typical assembly language programs. The discussion of procedures and functions continues in the next chapter; that chapter discusses advanced techniques that you will not commonly use in assembly language programs. The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- Procedures.
- ❑ Near and far procedures.
- Functions
- Saving the state of the machine
- Parameters
- Pass by value parameters.
- Pass by reference parameters.
- ❑ Pass by value-returned parameters.
- ❑ Pass by result parameters.
- ❑ Pass by name parameters.
- Passing parameters in registers.
- Passing parameters in global variables.
- Passing parameters on the stack.
- Passing parameters in the code stream.
- ❑ Passing parameters via a parameter block.
- Function results.
- Returning function results in a register.
- Returning function results on the stack.
- Returning function results in memory locations.
- Side effects.
- ❑ Local variable storage.
- ❑ Recursion.

11.1 Procedures

In a procedural environment, the basic unit of code is the *procedure*. A procedure is a set of instructions that compute some value or take some action (such as printing or reading a character value). The definition of a procedure is very similar to the definition of an *algorithm*. A procedure is a set of rules to follow which, if they conclude, produce some result. An algorithm is also such a sequence, but an algorithm is guaranteed to terminate whereas a procedure offers no such guarantee.

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86's *procedure invocation mechanism*. The calling code calls a procedure with the call instruction, the procedure returns to the caller with the ret instruction. For example, the following 80x86 instruction calls the UCR Standard Library sl_putcr routine[1]:

```
                call    sl_putcr
```

sl_putcr prints a carriage return/line feed sequence to the video display and returns control to the instruction immediately following the call sl_putcr instruction.

Alas, the UCR Standard Library does not supply all the routines you will need. Most of the time you'll have to write your own procedures. A simple procedure may consist of nothing more than a sequence of instructions ending with a ret instruction. For example, the following "procedure" zeros out the 256 bytes starting at the address in the bx register:

```
ZeroBytes:      xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
```

By loading the bx register with the address of some block of 256 bytes and issuing a call ZeroBytes instruction, you can zero out the specified block.

As a general rule, you won't define your own procedures in this manner. Instead, you should use MASM's proc and endp assembler directives. The ZeroBytes routine, using the proc and endp directives, is

```
ZeroBytes       proc
                xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
ZeroBytes       endp
```

Keep in mind that proc and endp are assembler directives. They do not generate any code. They're simply a mechanism to help make your programs easier to read. To the 80x86, the last two examples are identical; however, to a human being, latter is clearly a self-contained procedure, the other could simply be an arbitrary set of instructions within some other procedure. Consider now the following code:

```
ZeroBytes:      xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret

DoFFs:          mov     cx, 128
                mov     ax, 0ffffh
```

1. Normally you would use the putcr macro to accomplish this, but this call instruction will accomplish the same thing.

```
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
```

Are there two procedures here or just one? In other words, can a calling program enter this code at labels ZeroBytes and DoFFs or just at ZeroBytes? The use of the proc and endp directives can help remove this ambiguity:

Treated as a single subroutine:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret

DoFFs:          mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
ZeroBytes       endp
```

Treated as two separate routines:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
ZeroBytes       endp

DoFFs           proc
                mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
DoFFs           endp
```

Always keep in mind that the proc and endp directives are *logical* procedure separators. The 80x86 microprocessor returns from a procedure by executing a ret instruction, not by encountering an endp directive. The following is not equivalent to the code above:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
;       Note missing RET instr.
ZeroBytes       endp

DoFFs           proc
                mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
;       Note missing RET instr.
DoFFs           endp
```

Without the ret instruction at the end of each procedure, the 80x86 will fall into the next subroutine rather than return to the caller. After executing ZeroBytes above, the 80x86 will drop through to the DoFFs subroutine (beginning with the mov cx, 128 instruction).

Once DoFFs is through, the 80x86 will continue execution with the next executable instruction following DoFFs' endp directive.

An 80x86 procedure takes the form:

```
ProcName        proc     {near|far}   ;Choose near, far, or neither.
      <Procedure instructions>
ProcName        endp
```

The near or far operand is optional, the next section will discuss its purpose. The procedure name must be on the both proc and endp lines. The procedure name must be unique in the program.

Every proc directive must have a matching endp directive. Failure to match the proc and endp directives will produce a *block nesting error.*

11.2 Near and Far Procedures

The 80x86 supports near and far subroutines. Near calls and returns transfer control between procedures in the same code segment. Far calls and returns pass control between different segments. The two calling and return mechanisms push and pop different return addresses. You generally do not use a near call instruction to call a far procedure or a far call instruction to call a near procedure. Given this little rule, the next question is "how do you control the emission of a near or far call or ret?"

Most of the time, the call instruction uses the following syntax:

```
                call     ProcName
```

and the ret instruction is either[2]:

```
                ret
or              ret      disp
```

Unfortunately, these instructions do not tell MASM if you are calling a near or far procedure or if you are returning from a near or far procedure. The proc directive handles that chore. The proc directive has an optional operand that is either near or far. Near is the default if the operand field is empty[3]. The assembler assigns the procedure type (near or far) to the symbol. Whenever MASM assembles a call instruction, it emits a near or far call depending on operand. Therefore, declaring a symbol with proc or proc near, forces a near call. Likewise, using proc far, forces a far call.

Besides controlling the generation of a near or far call, proc's operand also controls ret code generation. If a procedure has the near operand, then all return instructions inside that procedure will be near. MASM emits far returns inside far procedures.

11.2.1 Forcing NEAR or FAR CALLs and Returns

Once in a while you might want to override the near/far declaration mechanism. MASM provides a mechanism that allows you to force the use of near/far calls and returns.

Use the near ptr and far ptr operators to override the automatic assignment of a near or far call. If NearLbl is a near label and FarLbl is a far label, then the following call instructions generate a near and far call, respectively:

```
                call     NearLbl                ;Generates a NEAR call.
                call     FarLbl                 ;Generates a FAR call.
```

Suppose you need to make a far call to NearLbl or a near call to FarLbl. You can accomplish this using the following instructions:

2. There are also retn and retf instructions.
3. Unless you are using MASM's *simplified segment directives.* See the appendices for details.

```
                call    far ptr NearLbl         ;Generates a FAR call.
                call    near ptr FarLbl         ;Generates a NEAR call.
```

Calling a near procedure using a far call, or calling a far procedure using a near call isn't something you'll normally do. If you call a near procedure using a far call instruction, the near return will leave the cs value on the stack. Generally, rather than:

```
                call    far ptr NearProc
```

you should probably use the clearer code:

```
                push    cs
                call    NearProc
```

Calling a far procedure with a near call is a very dangerous operation. If you attempt such a call, the current cs value must be on the stack. Remember, a far ret pops a segmented return address off the stack. A near call instruction only pushes the offset, not the segment portion of the return address.

Starting with MASM v5.0, there are explicit instructions you can use to force a near or far ret. If ret appears within a procedure declared via proc and end;, MASM will automatically generate the appropriate near or far return instruction. To accomplish this, use the retn and retf instructions. These two instructions generate a near and far ret, respectively.

11.2.2 Nested Procedures

MASM allows you to nest procedures. That is, one procedure definition may be totally enclosed inside another. The following is an example of such a pair of procedures:

```
OutsideProc     proc    near
                jmp     EndofOutside

InsideProc      proc    near
                mov     ax, 0
                ret
InsideProc      endp

EndofOutside:   call    InsideProc
                mov     bx, 0
                ret
OutsideProc     endp
```

Unlike some high level languages, nesting procedures in 80x86 assembly language doesn't serve any useful purpose. If you nest a procedure (as with InsideProc above), you'll have to code an explicit jmp around the nested procedure. Placing the nested procedure after all the code in the outside procedure (but still between the outside proc/endp directives) doesn't accomplish anything. Therefore, there isn't a good reason to nest procedures in this manner.

Whenever you nest one procedure within another, it must be totally contained within the nesting procedure. That is, the proc and endp statements for the nested procedure must lie between the proc and endp directives of the outside, nesting, procedure. The following is *not* legal:

```
OutsideProc     proc    near
                  .
                  .
                  .
InsideProc      proc    near
                  .
                  .
                  .
OutsideProc     endp
                  .
                  .
                  .
InsideProc      endp
```

The OutsideProc and InsideProc procedures overlap, they are not nested. If you attempt to create a set of procedures like this, MASM would report a "block nesting error". Figure 11.1 demonstrates this graphically.

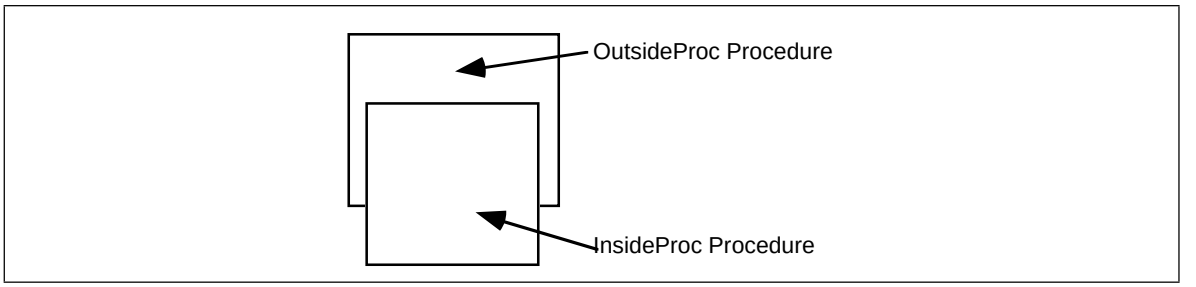


Figure 11.1 Illegal Procedure Nesting

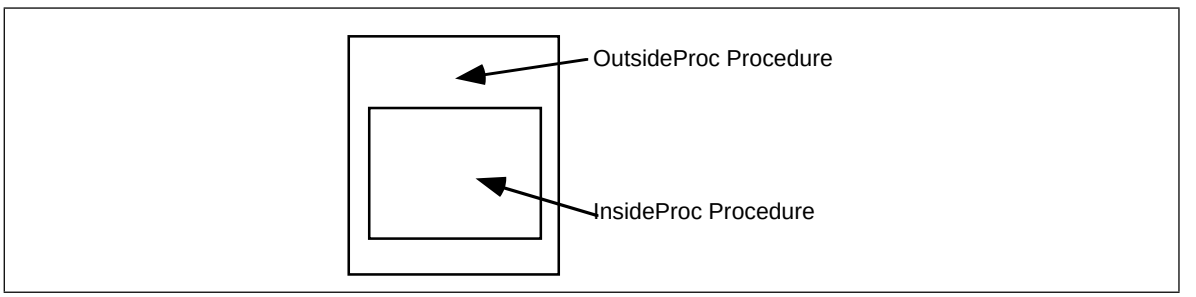


Figure 11.2 Legal Procedure Nesting

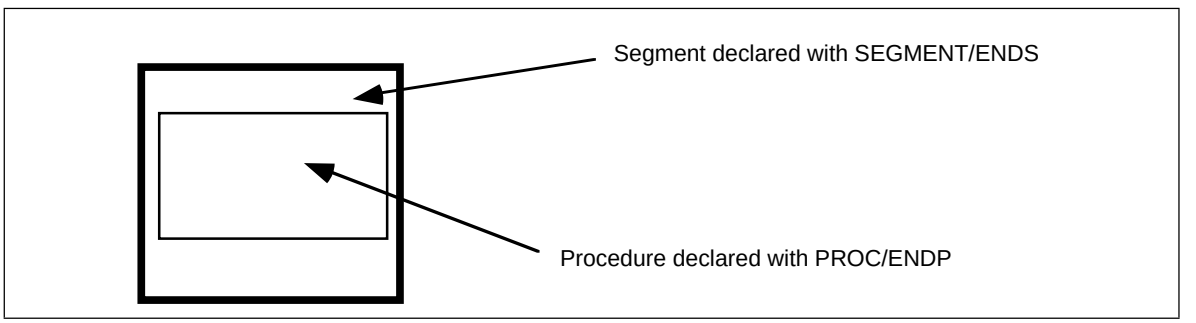


Figure 11.3 Legal Procedure/Segment Nesting

The only form acceptable to MASM appears in Figure 11.2.

Besides fitting inside an enclosing procedure, proc/endp groups must fit entirely within a segment. Therefore the following code is illegal:

```
cseg            segment
MyProc          proc    near
                ret
cseg            ends
MyProc          endp
```

The endp directive must appear before the cseg ends statement since MyProc begins inside cseg. Therefore, procedures within segments must always take the form shown in Figure 11.3.

Not only can you nest procedures inside other procedures and segments, but you can nest segments inside other procedures and segments as well. If you're the type who likes to simulate Pascal or C procedures in assembly language, you can create variable declaration sections at the beginning of each procedure you create, just like Pascal:

```
cgroup          group   cseg1, cseg2

cseg1           segment para public 'code'
cseg1           ends

cseg2           segment   para public 'code'
cseg2           ends
```

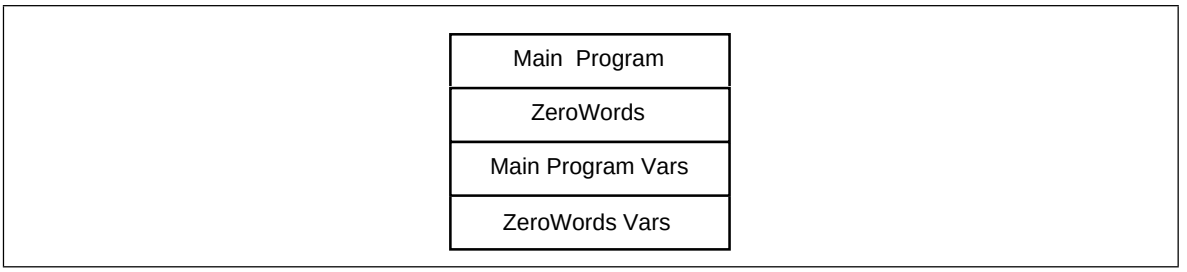


Figure 11.4 Example Memory Layout

```
dseg            segment     para public 'data'
dseg            ends

cseg1           segment     para public 'code'
                assume      cs:cgroup, ds:dseg

MainPgm         proc        near

; Data declarations for main program:

dseg            segment     para public 'data'
I               word        ?
J               word        ?
dseg            ends

; Procedures that are local to the main program:

cseg2           segment     para public 'code'

ZeroWords       proc        near

; Variables local to ZeroBytes:

dseg            segment     para public 'data'
AXSave          word        ?
BXSave          word        ?
CXSave          word        ?
dseg            ends

; Code for the ZeroBytes procedure:

                mov         AXSave, ax
                mov         CXSave, cx
                mov         BXSave, bx
                xor         ax, ax
ZeroLoop:       mov         [bx], ax
                inc         bx
                inc         bx
                loop        ZeroLoop
                mov         ax, AXSave
                mov         bx, BXSave
                mov         cx, CXSave
                ret
ZeroWords       endp

Cseg2           ends

; The actual main program begins here:

                mov         bx, offset Array
                mov         cx, 128
                call        ZeroWords
                ret
MainPgm         endp
cseg1           ends
                end
```

The system will load this code into memory as shown in Figure 11.4.

ZeroWords *follows* the main program because it belongs to a different segment (cseg2) than MainPgm (cseg1). Remember, the assembler and linker combine segments with the

same class name into a single segment before loading them into memory (see “Segment Loading Order” on page 368 for more details). You can use this feature of the assembler to “pseudo-Pascalize” your code in the fashion shown above. However, you’ll probably not find your programs to be any more readable than using the straight forward non-nesting approach.

11.3 Functions

The difference between functions and procedures in assembly language is mainly a matter of definition. The purpose for a function is to return some explicit value while the purpose for a procedure is to execute some action. To declare a function in assembly language, use the proc/endp directives. All the rules and techniques that apply to procedures apply to functions. This text will take another look at functions later in this chapter in the section on function results. From here on, procedure will mean procedure or function.

11.4 Saving the State of the Machine

Take a look at this code:

```
                mov     cx, 10
Loop0:          call    PrintSpaces
                putcr
                loop    Loop0
                  .
                  .
                  .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

This section of code attempts to print ten lines of 40 spaces each. Unfortunately, there is a subtle bug that causes it to print 40 spaces per line in an infinite loop. The main program uses the loop instruction to call PrintSpaces 10 times. PrintSpaces uses cx to count off the 40 spaces it prints. PrintSpaces returns with cx containing zero. The main program then prints a carriage return/line feed, decrements cx, and then repeats because cx isn’t zero (it will always contain 0FFFFh at this point).

The problem here is that the PrintSpaces subroutine doesn’t preserve the cx register. Preserving a register means you save it upon entry into the subroutine and restore it before leaving. Had the PrintSpaces subroutine preserved the contents of the cx register, the program above would have functioned properly.

Use the 80x86’s push and pop instructions to preserve register values while you need to use them for something else. Consider the following code for PrintSpaces:

```
PrintSpaces     proc    near
                push    ax
                push    cx
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                pop     cx
                pop     ax
                ret
PrintSpaces     endp
```

Note that PrintSpaces saves and restores ax and cx (since this procedure modifies these registers). Also, note that this code pops the registers off the stack in the reverse order that it pushed them. The operation of the stack imposes this ordering.

Either the caller (the code containing the call instruction) or the callee (the subroutine) can take responsibility for preserving the registers. In the example above, the callee preserved the registers. The following example shows what this code might look like if the caller preserves the registers:

```
                mov     cx, 10
Loop0:          push    ax
                push    cx
                call    PrintSpaces
                pop     cx
                pop     ax
                putcr
                loop    Loop0
                  .
                  .
                  .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

There are two advantages to callee preservation: space and maintainability. If the callee preserves all affected registers, then there is only one copy of the push and pop instructions, those the procedure contains. If the caller saves the values in the registers, the program needs a set of push and pop instructions around every call. Not only does this make your programs longer, it also makes them harder to maintain. Remembering which registers to push and pop on each procedure call is not something easily done.

On the other hand, a subroutine may unnecessarily preserve some registers if it preserves all the registers it modifies. In the examples above, the code needn't save ax. Although PrintSpaces changes the al, this won't affect the program's operation. If the caller is preserving the registers, it doesn't have to save registers it doesn't care about:

```
                mov     cx, 10
Loop0:          push    cx
                call    PrintSpaces
                pop     cx
                putcr
                loop    Loop0
                putcr
                putcr
                call    PrintSpaces

                mov     al, '*'
                mov     cx, 100
Loop1:          putc
                push    ax
                push    cx
                call    PrintSpaces
                pop     cx
                pop     ax
                putc
                putcr
                loop    Loop1
                  .
                  .
                  .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

This example provides three different cases. The first loop (Loop0) only preserves the cx register. Modifying the al register won't affect the operation of this program. Immediately after the first loop, this code calls PrintSpaces again. However, this code doesn't save

ax or cx because it doesn't care if PrintSpaces changes them. Since the final loop (Loop1) uses ax and cx, it saves them both.

One big problem with having the caller preserve registers is that your program may change. You may modify the calling code or the procedure so that they use additional registers. Such changes, of course, may change the set of registers that you must preserve. Worse still, if the modification is in the subroutine itself, you will need to locate *every* call to the routine and verify that the subroutine does not change any registers the calling code uses.

Preserving registers isn't all there is to preserving the environment. You can also push and pop variables and other values that a subroutine might change. Since the 80x86 allows you to push and pop memory locations, you can easily preserve these values as well.

11.5 Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. There are many facets to parameters. Questions concerning parameters include:

- *where* is the data coming from?
- *how* do you pass and return data?
- *what* is the amount of data to pass?

There are six major mechanisms for passing data to and from a procedure, they are

- pass by value,
- pass by reference,
- pass by value/returned,
- pass by result, and
- pass by name.
- pass by lazy evaluation

You also have to worry about where you can pass parameters. Common places are

- in registers,
- in global memory locations,
- on the stack,
- in the code stream, or
- in a parameter block referenced via a pointer.

Finally, the amount of data has a direct bearing on where and how to pass it. The following sections take up these issues.

11.5.1 Pass by Value

A parameter passed by value is just that – the caller passes a value to the procedure. Pass by value parameters are input only parameters. That is, you can pass them to a procedure but the procedure cannot return them. In HLLs, like Pascal, the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the Pascal procedure call:

```
CallProc(I);
```

If you pass I by value, the CallProc does not change the value of I, regardless of what happens to the parameter inside CallProc.

Since you must pass a copy of the data to the procedure, you should only use this method for passing small objects like bytes, words, and double words. Passing arrays and

strings by value is very inefficient (since you must create and pass a copy of the structure to the procedure).

11.5.2 Pass by Reference

To pass a parameter by reference, you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures.

Passing parameters by reference can produce some peculiar results. The following Pascal procedure provides an example of one problem you might encounter:

```
program main(input,output);
var m:integer;

                procedure bletch(var i,j:integer);
                begin

                        i := i+2;
                        j := j-i;
                        writeln(i,' ',j);

                end;

                  .
                  .
                  .
    begin {main}

                m := 5;
                bletch(m,m);

    end.
```

This particular code sequence will print “00” regardless of m’s value. This is because the parameters i and j are pointers to the actual data and they both point at the same object. Therefore, the statement j:=j-i; always produces zero since i and j refer to the same variable.

Pass by reference is usually less efficient than pass by value. You must dereference all pass by reference parameters on each access; this is slower than simply using a value. However, when passing a large data structure, pass by reference is faster because you do not have to copy a large data structure before calling the procedure.

11.5.3 Pass by Value-Returned

Pass by value-returned (also known as *value-result*) combines features from both the pass by value and pass by reference mechanisms. You pass a value-returned parameter by address, just like pass by reference parameters. However, upon entry, the procedure makes a temporary copy of this parameter and uses the copy while the procedure is executing. When the procedure finishes, it copies the temporary copy back to the original parameter.

The Pascal code presented in the previous section would operate properly with pass by value-returned parameters. Of course, when Bletch returns to the calling code, m could only contain one of the two values, but while Bletch is executing, i and j would contain distinct values.

In some instances, pass by value-returned is more efficient than pass by reference, in others it is less efficient. If a procedure only references the parameter a couple of times, copying the parameter’s data is expensive. On the other hand, if the procedure uses this parameter often, the procedure amortizes the fixed cost of copying the data over many inexpensive accesses to the local copy.

11.5.4 Pass by Result

Pass by result is almost identical to pass by value-returned. You pass in a pointer to the desired object and the procedure uses a local copy of the variable and then stores the result through the pointer when returning. The only difference between pass by value-returned and pass by result is that when passing parameters by result you do not copy the data upon entering the procedure. Pass by result parameters are for returning values, not passing data to the procedure. Therefore, pass by result is slightly more efficient than pass by value-returned since you save the cost of copying the data into the local variable.

11.5.5 Pass by Name

Pass by name is the parameter passing mechanism used by macros, text equates, and the #define macro facility in the C programming language. This parameter passing mechanism uses textual substitution on the parameters. Consider the following MASM macro:

```
PassByName      macro   Parameter1, Parameter2
                mov     ax, Parameter1
                add     ax, Parameter2
                endm
```

If you have a macro invocation of the form:

```
                PassByName  bx, I
```

MASM emits the following code, substituting bx for Parameter1 and I for Parameter2:

```
                mov     ax, bx
                add     ax, I
```

Some high level languages, such as ALGOL-68 and Panacea, support pass by name parameters. However, implementing pass by name using textual substitution in a compiled language (like ALGOL-68) is very difficult and inefficient. Basically, you would have to recompile a function everytime you call it. So compiled languages that support pass by name parameters generally use a different technique to pass those parameters. Consider the following Panacea procedure:

```
PassByName: procedure(name item:integer; var index:integer);
begin PassByName;

        foreach index in 0..10 do

                item := 0;

        endfor;

end PassByName;
```

Assume you call this routine with the statement PassByName(A[i], i); where A is an array of integers having (at least) the elements A[0]..A[10]. Were you to substitute the pass by name parameter *item* you would obtain the following code:

```
begin PassByName;

        foreach index in 0..10 do

                A[I] := 0; (* Note that index and I are aliases *)

        endfor;

end PassByName;
```

This code zeros out elements 0..10 of array A.

High level languages like ALGOL-68 and Panacea compile pass by name parameters into *functions* that return the address of a given parameter. So in one respect, pass by name parameters are similar to pass by reference parameters insofar as you pass the address of an object. The major difference is that with pass by reference you compute the

address of an object before calling a subroutine; with pass by name the subroutine itself calls some function to compute the address of the parameter.

So what difference does this make? Well, reconsider the code above. Had you passed A[I] by reference rather than by name, the calling code would compute the address of A[I] *just before the call* and passed in this address. Inside the PassByName procedure the variable item would have always referred to a single address, not an address that changes along with I. With pass by name parameters, item is really a function that computes the address of the parameter into which the procedure stores the value zero. Such a function might look like the following:

```
ItemThunk       proc    near
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
ItemThunk       endp
```

The compiled code inside the PassByName procedure might look something like the following:

```
; item := 0;

                call    ItemThunk
                mov     word ptr [bx], 0
```

Thunk is the historical term for these functions that compute the address of a pass by name parameter. It is worth noting that most HLLs supporting pass by name parameters do not call thunks directly (like the call above). Generally, the caller passes the address of a thunk and the subroutine calls the thunk *indirectly*. This allows the same sequence of instructions to call several different thunks (corresponding to different calls to the subroutine).

11.5.6 Pass by Lazy-Evaluation

Pass by name is similar to pass by reference insofar as the procedure accesses the parameter using the address of the parameter. The primary difference between the two is that a caller directly passes the address on the stack when passing by reference, it passes the address of a function that computes the parameter's address when passing a parameter by name. The pass by lazy evaluation mechanism shares this same relationship with pass by value parameters – the caller passes the address of a function that computes the parameter's value if the first access to that parameter is a read operation.

Pass by lazy evaluation is a useful parameter passing technique if the cost of computing the parameter value is very high and the procedure may not use the value. Consider the following Panacea procedure header:

```
PassByEval: procedure(eval a:integer; eval b:integer; eval c:integer);
```

When you call the PassByEval function it does not evaluate the actual parameters and pass their values to the procedure. Instead, the compiler generates thunks that will compute the value of the parameter at most one time. If the first access to an eval parameter is a read, the thunk will compute the parameter's value and store that into a local variable. It will also set a flag so that all future accesses will not call the thunk (since it has already computed the parameter's value). If the first access to an eval parameter is a write, then the code sets the flag and future accesses within the same procedure activation will use the written value and ignore the thunk.

Consider the PassByEval procedure above. Suppose it takes several minutes to compute the values for the a, b, and c parameters (these could be, for example, three different possible paths in a Chess game). Perhaps the PassByEval procedure only uses the value of one of these parameters. Without pass by lazy evaluation, the calling code would have to spend the time to compute all three parameters even though the procedure will only use one of the values. With pass by lazy evaluation, however, the procedure will only spend

the time computing the value of the one parameter it needs. Lazy evaluation is a common technique artificial intelligence (AI) and operating systems use to improve performance.

11.5.7 Passing Parameters in Registers

Having touched on how to pass parameters to a procedure, the next thing to discuss is where to pass parameters. Where you pass parameters depends, to a great extent, on the size and number of those parameters. If you are passing a small number of bytes to a procedure, then the registers are an excellent place to pass parameters. The registers are an ideal place to pass value parameters to a procedure. If you are passing a single parameter to a procedure you should use the following registers for the accompanying data types:

Data Size	Pass in this Register
Byte:	al
Word:	ax
Double Word:	dx:ax or eax (if 80386 or better)

This is, by no means, a hard and fast rule. If you find it more convenient to pass 16 bit values in the si or bx register, by all means do so. However, most programmers use the registers above to pass parameters.

If you are passing several parameters to a procedure in the 80x86's registers, you should probably use up the registers in the following order:

First Last

ax, dx, si, di, bx, cx

In general, you should avoid using bp register. If you need more than six words, perhaps you should pass your values elsewhere.

The UCR Standard Library package provides several good examples of procedures that pass parameters by value in the registers. Putc, which outputs an ASCII character code to the video display, expects an ASCII value in the al register. Likewise, puti expects the value of a signed integer in the ax register. As another example, consider the following putsi (put short integer) routine that outputs the value in al as a signed integer:

```
putsi           proc
                push    ax                      ;Save AH's value.
                cbw                             ;Sign extend AL -> AX.
                puti                            ;Let puti do the real work.
                pop     ax                      ;Restore AH.
                ret
putsi           endp
```

The other four parameter passing mechanisms (pass by reference, value-returned, result, and name) generally require that you pass a pointer to the desired object (or to a thunk in the case of pass by name). When passing such parameters in registers, you have to consider whether you're passing an offset or a full segmented address. Sixteen bit offsets can be passed in any of the 80x86's general purpose 16 bit registers. si, di, and bx are the best place to pass an offset since you'll probably need to load it into one of these registers anyway[4]. You can pass 32 bit segmented addresses dx:ax like other double word parameters. However, you can also pass them in ds:bx, ds:si, ds:di, es:bx, es:si, or es:di and be able to use them without copying into a segment register.

The UCR Stdlib routine puts, which prints a string to the video display, is a good example of a subroutine that uses pass by reference. It wants the address of a string in the es:di register pair. It passes the parameter in this fashion, not because it modifies the parameter, but because strings are rather long and passing them some other way would be inefficient. As another example, consider the following strfill(str,c) that copies the char-

4. This does not apply to thunks. You may pass the address of a thunk in any 16 bit register. Of course, on an 80386 or later processor, you can use any of the 80386's 32-bit registers to hold an address.

acter c (passed by value in al) to each character position in str (passed by reference in es:di) up to a zero terminating byte:

```
; strfill-      copies value in al to the string pointed at by es:di
;               up to a zero terminating byte.

byp             texequ  <byte ptr>

strfill         proc
                pushf                           ;Save direction flag.
                cld                             ;To increment D with STOS.
                push    di                      ;Save, because it's changed.
                jmp     sfStart

sfLoop:         stosb                           ;es:[di] := al, di := di + 1;
sfStart:        cmp     byp es:[di], 0          ;Done yet?
                jne     sfLoop

                pop     di                      ;Restore di.
                popf                            ;Restore direction flag.
                ret
strfill         endp
```

When passing parameters by value-returned or by result to a subroutine, you could pass in the address in a register. Inside the procedure you would copy the value pointed at by this register to a local variable (value-returned only). Just before the procedure returns to the caller, it could store the final result back to the address in the register.

The following code requires two parameters. The first is a pass by value-returned parameter and the subroutine expects the address of the actual parameter in bx. The second is a pass by result parameter whose address is in si. This routine increments the pass by value-result parameter and stores the previous result in the pass by result parameter:

```
; CopyAndInc-   BX contains the address of a variable. This routine
;               copies that variable to the location specified in SI
;               and then increments the variable BX points at.
;               Note: AX and CX hold the local copies of these
;               parameters during execution.

CopyAndInc      proc
                push    ax          ;Preserve AX across call.
                push    cx          ;Preserve CX across call.
                mov     ax, [bx]    ;Get local copy of 1st parameter.
                mov     cx, ax      ;Store into 2nd parm's local var.
                inc     ax          ;Increment 1st parameter.
                mov     [si], cx    ;Store away pass by result parm.
                mov     [bx], ax    ;Store away pass by value/ret parm.
                pop     cx          ;Restore CX's value.
                pop     ax          ;Restore AX's value.
                ret
CopyAndInc      endp
```

To make the call CopyAndInc(I,J) you would use code like the following:

```
                lea     bx, I
                lea     si, J
                call    CopyAndInc
```

This is, of course, a trivial example whose implementation is very inefficient. Nevertheless, it shows how to pass value-returned and result parameters in the 80x86's registers. If you are willing to trade a little space for some speed, there is another way to achieve the same results as pass by value-returned or pass by result when passing parameters in registers. Consider the following implementation of CopyAndInc:

```
CopyAndInc      proc
                mov     cx, ax      ;Make a copy of the 1st parameter,
                inc     ax          ; then increment it by one.
                ret
CopyAndInc      endp
```

To make the CopyAndInc(I,J) call, as before, you would use the following 80x86 code:

```
                mov     ax, I
                call    CopyAndInc
                mov     I, ax
                mov     J, cx
```

Note that this code does not pass any addresses at all; yet it has the same semantics (that is, performs the same operations) as the previous version. Both versions increment I and store the pre-incremented version into J. Clearly the latter version is faster, although your program will be slightly larger if there are many calls to CopyAndInc in your program (six or more).

You can pass a parameter by name or by lazy evaluation in a register by simply loading that register with the address of the thunk to call. Consider the Panacea PassByName procedure (see "Pass by Name" on page 576). One implementation of this procedure could be the following:

```
;PassByName-    Expects a pass by reference parameter index
;               passed in si and a pass by name parameter, item,
;               passed in dx (the thunk returns the address in bx).

PassByName      proc
                push    ax                      ;Preserve AX across call
                mov     word ptr [si], 0        ;Index := 0;
ForLoop:        cmp     word ptr [si], 10       ;For loop ends at ten.
                jg      ForDone
                call    dx                      ;Call thunk item.
                mov     word ptr [bx], 0        ;Store zero into item.
                inc     word ptr [si]           ;Index := Index + 1;
                jmp     ForLoop

ForDone:        pop     ax                      ;Restore AX.
                ret                             ;All Done!
PassByName      endp
```

You might call this routine with code that looks like the following:

```
                lea     si, I
                lea     dx, Thunk_A
                call    PassByName
                  .
                  .
                  .
Thunk_A         proc
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
Thunk_A         endp
```

The advantage to this scheme, over the one presented in the earlier section, is that you can call different thunks, not just the ItemThunk routine appearing in the earlier example.

11.5.8 Passing Parameters in Global Variables

Once you run out of registers, the only other (reasonable) alternative you have is main memory. One of the easiest places to pass parameters is in global variables in the data segment. The following code provides an example:

```
                mov     ax, xxxx                ;Pass this parameter by value
                mov     Value1Proc1, ax
                mov     ax, offset yyyy         ;Pass this parameter by ref
                mov     word ptr Ref1Proc1, ax
                mov     ax, seg yyyy
                mov     word ptr Ref1Proc1+2, ax
                call    ThisProc
                  .
                  .
                  .
```

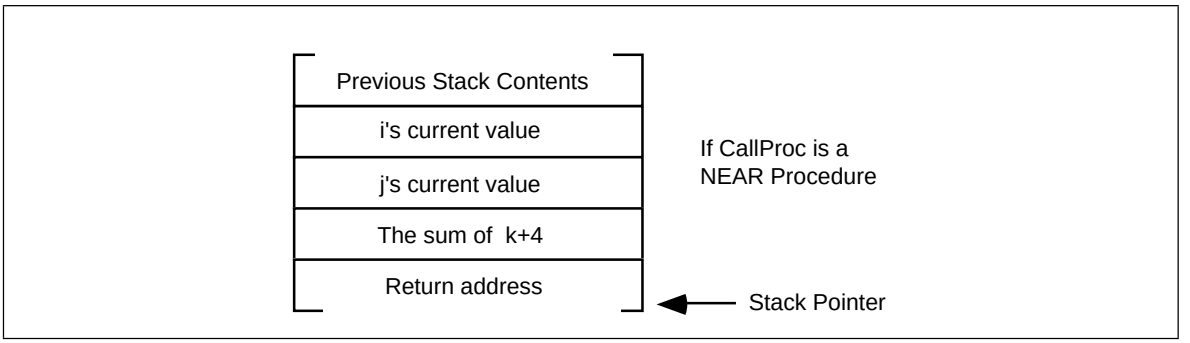


Figure 11.5 CallProc Stack Layout for a Near Procedure

```
ThisProc        proc    near
                push    es
                push    ax
                push    bx
                les     bx, Ref1Proc1           ;Get address of ref parm.
                mov     ax, Value1Proc1         ;Get value parameter
                mov     es:[bx], ax             ;Store into loc pointed at by
                pop     bx                      ; the ref parameter.
                pop     ax
                pop     es
                ret
ThisProc        endp
```

Passing parameters in global locations is inelegant and inefficient. Furthermore, if you use global variables in this fashion to pass parameters, the subroutines you write cannot use recursion (see “Recursion” on page 606). Fortunately, there are better parameter passing schemes for passing data in memory so you do not need to seriously consider this scheme.

11.5.9 Passing Parameters on the Stack

Most HLLs use the stack to pass parameters because this method is fairly efficient. To pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following Pascal procedure call:

```
CallProc(i,j,k+4);
```

Most Pascal compilers push their parameters onto the stack in the order that they appear in the parameter list. Therefore, the 80x86 code typically emitted for this subroutine call (assuming you’re passing the parameters by value) is

```
push    i
push    j
mov     ax, k
add     ax, 4
push    ax
call    CallProc
```

Upon entry into CallProc, the 80x86’s stack looks like that shown in Figure 11.5 (for a near procedure) or Figure 11.6 (for a far procedure).

You could gain access to the parameters passed on the stack by removing the data from the stack (Assuming a near procedure call):

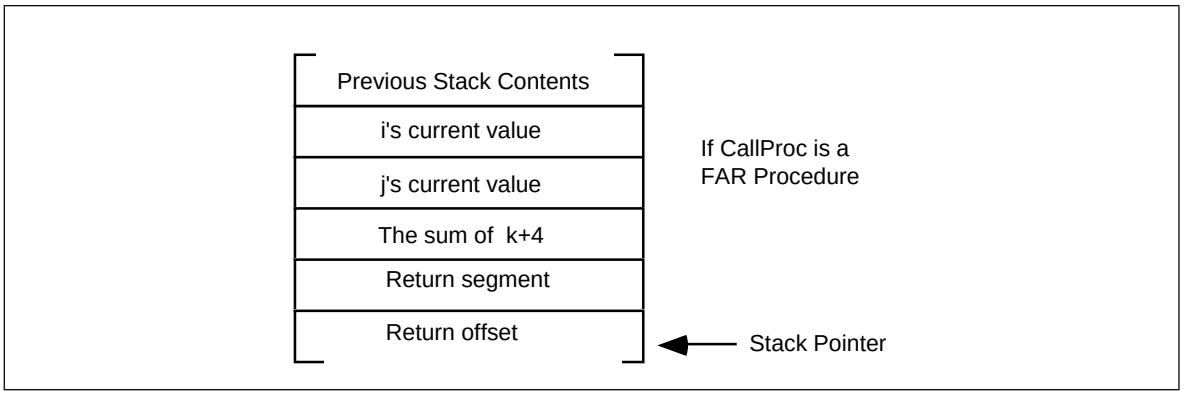


Figure 11.6 CallProc Stack Layout for a Far Procedure

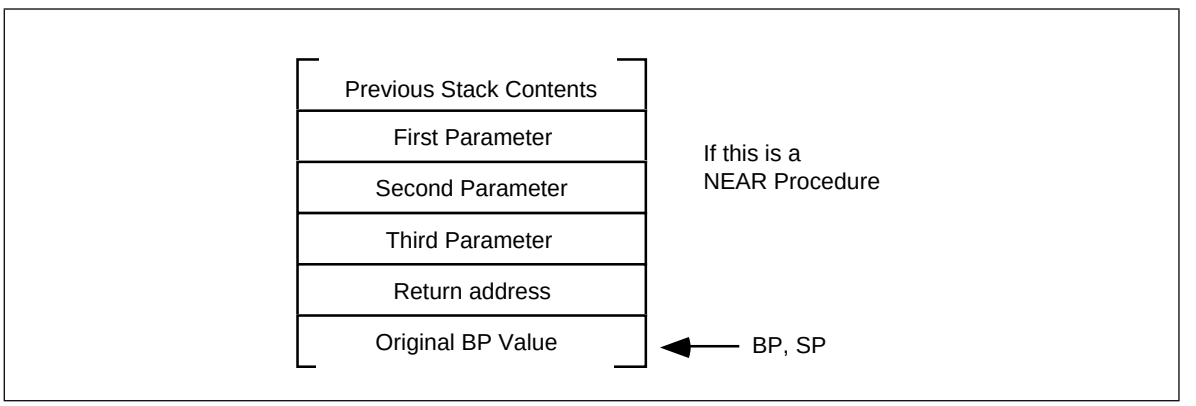


Figure 11.7 Accessing Parameters on the Stack

```
CallProc        proc    near
                pop     RtnAdrs
                pop     kParm
                pop     jParm
                pop     iParm
                push    RtnAdrs
                  .
                  .
                  .
                ret
CallProc        endp
```

There is, however, a better way. The 80x86's architecture allows you to use the bp (base pointer) register to access parameters passed on the stack. This is one of the reasons the disp[bp], [bp][di], [bp][si], disp[bp][si], and disp[bp][di] addressing modes use the stack segment rather than the data segment. The following code segment gives the *standard procedure entry and exit* code:

```
StdProc         proc    near
                push    bp
                mov     bp, sp
                  .
                  .
                  .
                pop     bp
                ret     ParmSize
StdProc         endp
```

ParmSize is the number of bytes of parameters pushed onto the stack before calling the procedure. In the CallProc procedure there were six bytes of parameters pushed onto the stack so ParmSize would be six.

Take a look at the stack immediately after the execution of mov bp, sp in StdProc. Assuming you've pushed three parameter words onto the stack, it should look something like shown in Figure 11.7.

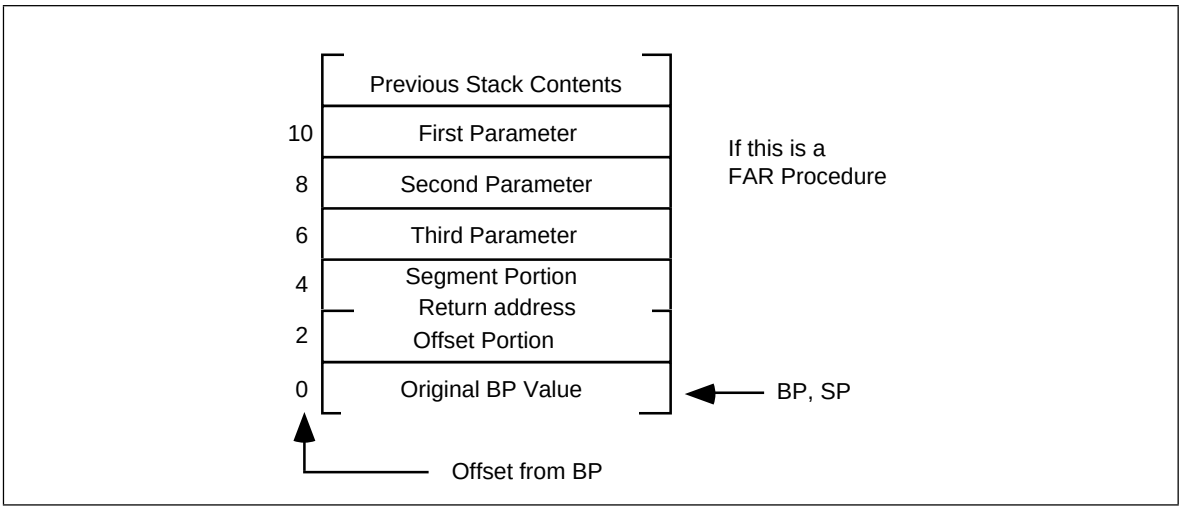


Figure 11.8 Accessing Parameters on the Stack in a Far Procedure

Now the parameters can be fetched by indexing off the bp register:

```
mov     ax, 8[bp]    ;Accesses the first parameter
mov     ax, 6[bp]    ;Accesses the second parameter
mov     ax, 4[bp]    ;Accesses the third parameter
```

When returning to the calling code, the procedure must remove these parameters from the stack. To accomplish this, pop the old bp value off the stack and execute a ret 6 instruction. This pops the return address off the stack and adds six to the stack pointer, effectively removing the parameters from the stack.

The displacements given above are for *near* procedures only. When calling a far procedure,

- 0[BP] will point at the old BP value,
- 2[BP] will point at the offset portion of the return address,
- 4[BP] will point at the segment portion of the return address, and
- 6[BP] will point at the last parameter pushed onto the stack.

The stack contents when calling a far procedure are shown in Figure 11.8.

This collection of parameters, return address, registers saved on the stack, and other items, is a *stack frame* or *activation record.*

When saving other registers onto the stack, always make sure that you save and set up bp before pushing the other registers. If you push the other registers before setting up bp, the offsets into the stack frame will change. For example, the following code disturbs the ordering presented above:

```
FunnyProc       proc    near
                push    ax
                push    bx
                push    bp
                mov     bp, sp
                  .
                  .
                  .
                pop     bp
                pop     bx
                pop     ax
                ret
FunnyProc       endp
```

Since this code pushes ax and bx before pushing bp and copying sp to bp, ax and bx appear in the activation record before the return address (that would normally start at location [bp+2]). As a result, the value of bx appears at location [bp+2] and the value of ax appears at location [bp+4]. This pushes the return address and other parameters farther up the stack as shown in Figure 11.9.

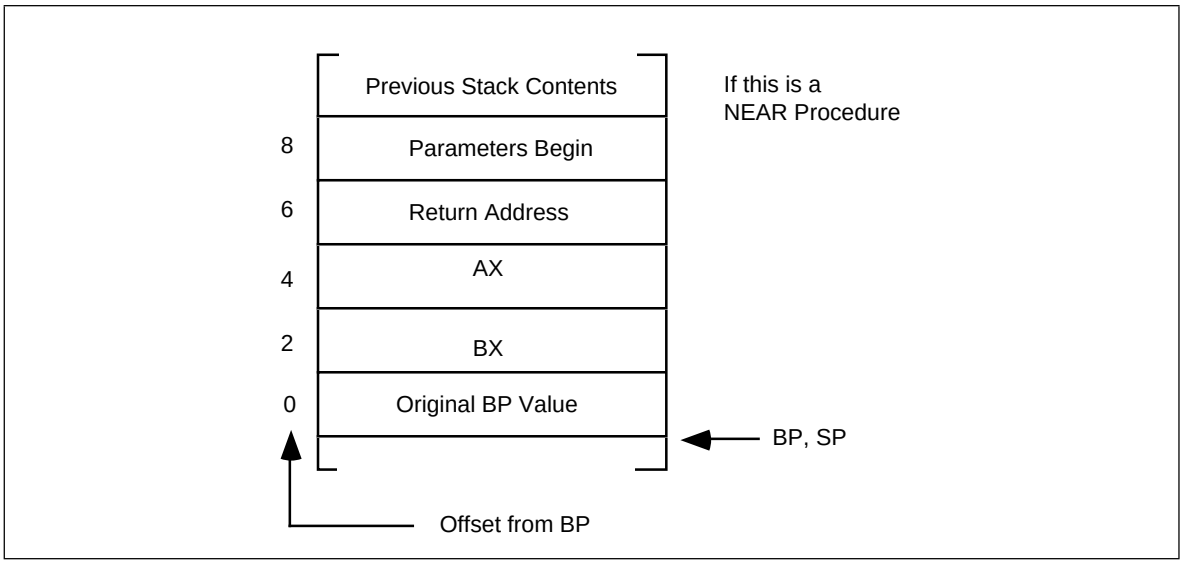


Figure 11.9 Messing up Offsets by Pushing Other Registers Before BP

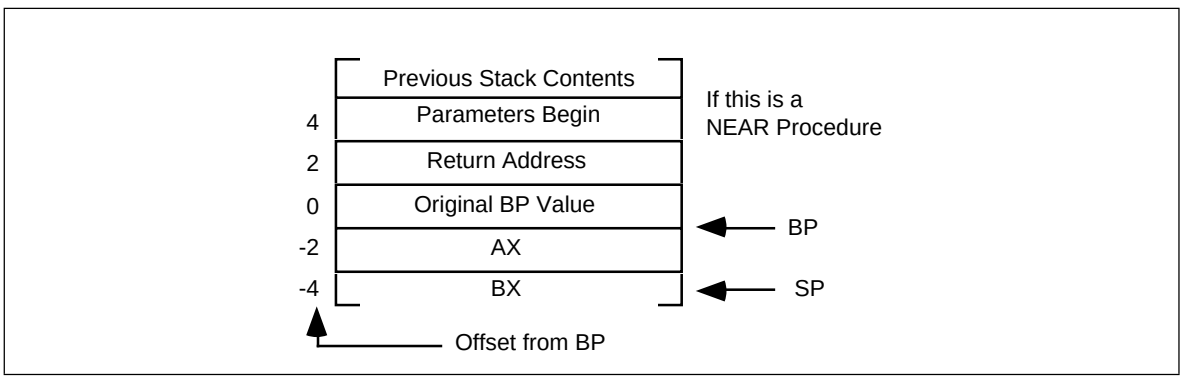


Figure 11.10 Keeping the Offsets Constant by Pushing BP First

Although this is a near procedure, the parameters don't begin until offset eight in the activation record. Had you pushed the ax and bx registers after setting up bp, the offset to the parameters would have been four (see Figure 11.10).

```
FunnyProc       proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                  .
                  .
                  .
                pop     bx
                pop     ax
                pop     bp
                ret
FunnyProc       endp
```

Therefore, the push bp and mov bp, sp instructions should be the first two instructions any subroutine executes when it has parameters on the stack.

Accessing the parameters using expressions like [bp+6] can make your programs very hard to read and maintain. If you would like to use meaningful names, there are several ways to do so. One way to reference parameters by name is to use equates. Consider the following Pascal procedure and its equivalent 80x86 assembly language code:

```
procedure xyz(var i:integer; j,k:integer);
begin
        i := j+k;
end;
```

Calling sequence:

```
        xyz(a,3,4);
```

Assembly language code:

```
xyz_i           equ     8[bp]       ;Use equates so we can reference
xyz_j           equ     6[bp]       ; symbolic names in the body of
xyz_k           equ     4[bp]       ; the procedure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    es
                push    ax
                push    bx
                les     bx, xyz_i   ;Get address of I into ES:BX
                mov     ax, xyz_j   ;Get J parameter
                add     ax, xyz_k   ;Add to K parameter
                mov     es:[bx], ax ;Store result into I parameter
                pop     bx
                pop     ax
                pop     es
                pop     bp
                ret     8
xyz             endp
```

Calling sequence:

```
                mov     ax, seg a           ;This parameter is passed by
                push    ax                  ; reference, so pass its
                mov     ax, offset a        ; address on the stack.
                push    ax
                mov     ax, 3               ;This is the second parameter
                push    ax
                mov     ax, 4               ;This is the third parameter.
                push    ax
                call    xyz
```

On an 80186 or later processor you could use the following code in place of the above:

```
                push    seg a               ;Pass address of "a" on the
                push    offset a            ; stack.
                push    3                   ;Pass second parm by val.
                push    4                   ;Pass third parm by val.
                call    xyz
```

Upon entry into the xyz procedure, before the execution of the les instruction, the stack looks like shown in Figure 11.11.

Since you're passing I by reference, you must push its address onto the stack. This code passes reference parameters using 32 bit segmented addresses. Note that this code uses ret 8. Although there are three parameters on the stack, the reference parameter I consumes four bytes since it is a far address. Therefore there are eight bytes of parameters on the stack necessitating the ret 8 instruction.

Were you to pass I by reference using a near pointer rather than a far pointer, the code would look like the following:

```
xyz_i           equ     8[bp]       ;Use equates so we can reference
xyz_j           equ     6[bp]       ; symbolic names in the body of
xyz_k           equ     4[bp]       ; the procedure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, xyz_i   ;Get address of I into BX
```

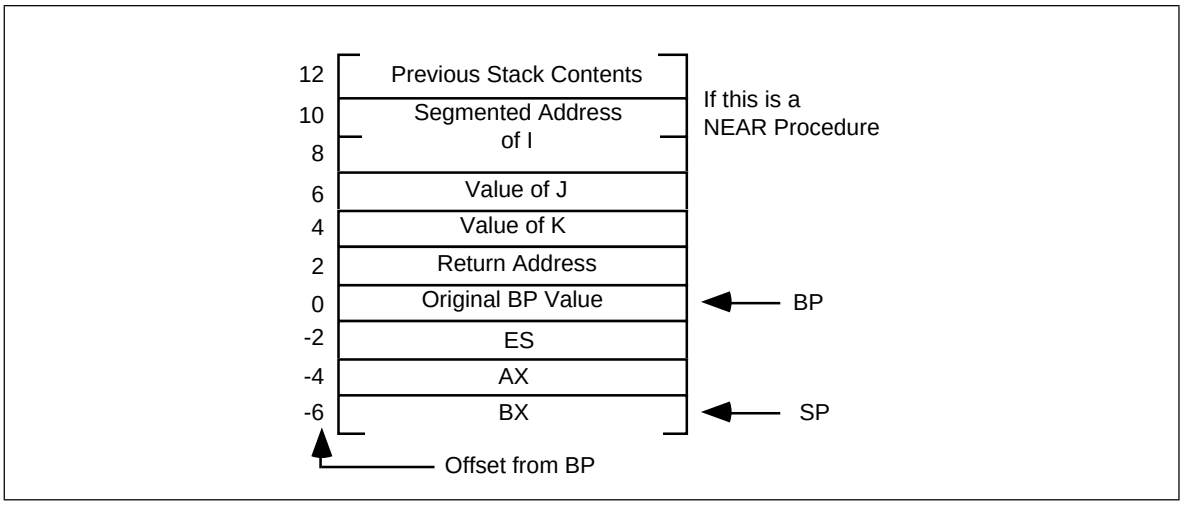


Figure 11.11 XYZ Stack Upon Procedure Entry

```
                mov     ax, xyz_j    ;Get J parameter
                add     ax, xyz_k    ;Add to K parameter
                mov     [bx], ax     ;Store result into I parameter
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

Note that since I's address on the stack is only two bytes (rather than four), this routine only pops six bytes when it returns.

Calling sequence:

```
                mov     ax, offset a ;Pass near address of a.
                push    ax
                mov     ax, 3        ;This is the second parameter
                push    ax
                mov     ax, 4        ;This is the third parameter.
                push    ax
                call    xyz
```

On an 80286 or later processor you could use the following code in place of the above:

```
                push    offset a             ;Pass near address of a.
                push    3                    ;Pass second parm by val.
                push    4                    ;Pass third parm by val.
                call    xyz
```

The stack frame for the above code appears in Figure 11.12.

When passing a parameter by value-returned or result, you pass an address to the procedure, exactly like passing the parameter by reference. The only difference is that you use a local copy of the variable within the procedure rather than accessing the variable indirectly through the pointer. The following implementations for xyz show how to pass I by value-returned and by result:

```
; xyz version using Pass by Value-Returned for xyz_i

xyz_i           equ     8[bp]        ;Use equates so we can reference
xyz_j           equ     6[bp]        ; symbolic names in the body of
xyz_k           equ     4[bp]        ; the procedure.

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
```

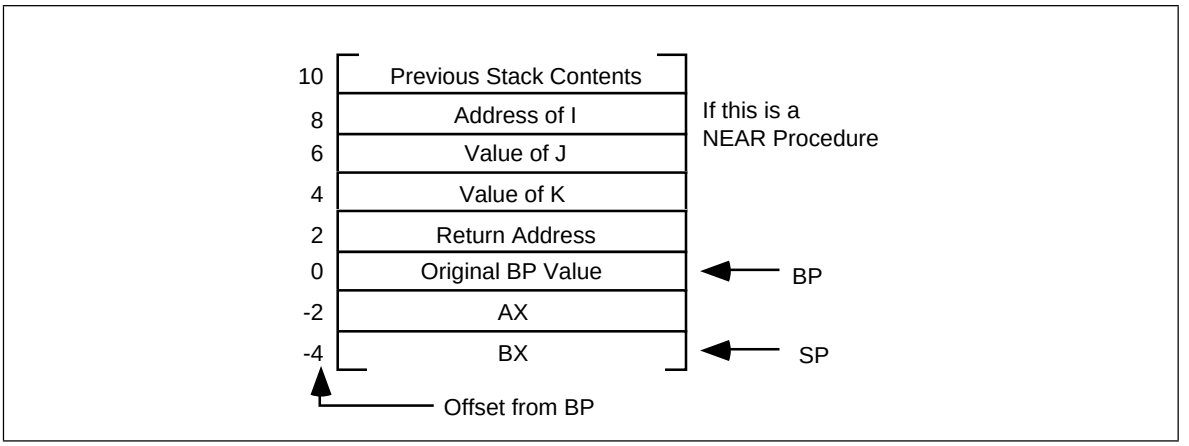


Figure 11.12 Passing Parameters by Reference Using Near Pointers Rather than Far Pointers

```
                        push    cx              ;Keep local copy here.

                        mov     bx, xyz_i       ;Get address of I into BX
                        mov     cx, [bx]        ;Get local copy of I parameter.

                        mov     ax, xyz_j       ;Get J parameter
                        add     ax, xyz_k       ;Add to K parameter
                        mov     cx, ax          ;Store result into local copy

                        mov     bx, xyz_i       ;Get ptr to I, again
                        mov     [bx], cx        ;Store result away.

                        pop     cx
                        pop     bx
                        pop     ax
                        pop     bp
                        ret     6
xyz                     endp
```

There are a couple of unnecessary mov instructions in this code. They are present only to precisely implement pass by value-returned parameters. It is easy to improve this code using pass by result parameters. The modified code is

```
; xyz version using Pass by Result for xyz_i

xyz_i                   equ     8[bp]           ;Use equates so we can reference
xyz_j                   equ     6[bp]           ; symbolic names in the body of
xyz_k                   equ     4[bp]           ; the procedure.

xyz                     proc    near
                        push    bp
                        mov     bp, sp
                        push    ax
                        push    bx
                        push    cx              ;Keep local copy here.

                        mov     ax, xyz_j       ;Get J parameter
                        add     ax, xyz_k       ;Add to K parameter
                        mov     cx, ax          ;Store result into local copy

                        mov     bx, xyz_i       ;Get ptr to I, again
                        mov     [bx], cx        ;Store result away.

                        pop     cx
                        pop     bx
                        pop     ax
                        pop     bp
                        ret     6
xyz                     endp
```

As with passing value-returned and result parameters in registers, you can improve the performance of this code using a modified form of pass by value. Consider the following implementation of xyz:

```
; xyz version using modified pass by value-result for xyz_i

xyz_i           equ     8[bp]       ;Use equates so we can reference
xyz_j           equ     6[bp]       ; symbolic names in the body of
xyz_k           equ     4[bp]       ; the procedure.

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax

                mov     ax, xyz_j   ;Get J parameter
                add     ax, xyz_k   ;Add to K parameter
                mov     xyz_i, ax   ;Store result into local copy

                pop     ax
                pop     bp
                ret     4           ;Note that we do not pop I parm.
xyz             endp
```

The calling sequence for this code is

```
                push    a           ;Pass a's value to xyz.
                push    3           ;Pass second parameter by val.
                push    4           ;Pass third parameter by val.
                call    xyz
                pop     a
```

Note that a pass by result version wouldn't be practical since you have to push *something* on the stack to make room for the local copy of `I` inside xyz. You may as well push the value of `a` on entry even though the xyz procedure ignores it. This procedure pops only *four* bytes off the stack on exit. This leaves the value of the `I` parameter on the stack so that the calling code can store it away to the proper destination.

To pass a parameter by name on the stack, you simply push the address of the thunk. Consider the following pseudo-Pascal code:

```
procedure swap(name Item1, Item2:integer);
var temp:integer;
begin

       temp := Item1;
       Item1 := Item2;
       Item2 := Temp;

end;
```

If swap is a near procedure, the 80x86 code for this procedure could look like the following (note that this code has been slightly optimized and does not following the exact sequence given above):

```
; swap-         swaps two parameters passed by name on the stack.
;               Item1 is passed at address [bp+6], Item2 is passed
;               at address [bp+4]

wp              textequ  <word ptr>
swap_Item1      equ      [bp+6]
swap_Item2      equ      [bp+4]

swap            proc     near
                push     bp
                mov      bp, sp
                push     ax                     ;Preserve temp value.
                push     bx                     ;Preserve bx.
                call     wp swap_Item1          ;Get adrs of Item1.
                mov      ax, [bx]               ;Save in temp (AX).
                call     wp swap_Item2          ;Get adrs of Item2.
                xchg     ax, [bx]               ;Swap temp <-> Item2.
                call     wp swap_Item1          ;Get adrs of Item1.
```

```
                mov     [bx], ax                ;Save temp in Item1.
                pop     bx                      ;Restore bx.
                pop     ax                      ;Restore ax.
                ret     4                       ;Return and pop Item1/2.
swap            endp
```

Some sample calls to swap follow:

```
; swap(A[i], i) -- 8086 version.

                lea     ax, thunk1
                push    ax
                lea     ax, thunk2
                push    ax
                call    swap

; swap(A[i],i) -- 80186 & later version.

                push    offset thunk1
                push    offset thunk2
                call    swap

                 .
                 .
                 .

; Note: this code assumes A is an array of two byte integers.

thunk1          proc    near
                mov     bx, i
                shl     bx, 1
                lea     bx, A[bx]
                ret
thunk1          endp

thunk2          proc    near
                lea     bx, i
                ret
thunk2          endp
```

The code above assumes that the thunks are near procs that reside in the same segment as the swap routine. If the thunks are far procedures the caller must pass far addresses on the stack and the swap routine must manipulate far addresses. The following implementation of swap, thunk1, and thunk2 demonstrate this.

```
; swap-         swaps two parameters passed by name on the stack.
;               Item1 is passed at address [bp+10], Item2 is passed
;               at address [bp+6]

swap_Item1      equ     [bp+10]
swap_Item2      equ     [bp+6]
dp              textequ <dword ptr>

swap            proc    far
                push    bp
                mov     bp, sp
                push    ax                      ;Preserve temp value.
                push    bx                      ;Preserve bx.
                push    es                      ;Preserve es.
                call    dp swap_Item1           ;Get adrs of Item1.
                mov     ax, es:[bx]             ;Save in temp (AX).
                call    dp swap_Item2           ;Get adrs of Item2.
                xchg    ax, es:[bx]             ;Swap temp <-> Item2.
                call    dp swap_Item1           ;Get adrs of Item1.
                mov     es:[bx], ax             ;Save temp in Item1.
                pop     es                      ;Restore es.
                pop     bx                      ;Restore bx.
                pop     ax                      ;Restore ax.
                ret     8                       ;Return and pop Item1, Item2.
swap            endp
```

Some sample calls to swap follow:

```
; swap(A[i], i) -- 8086 version.

                mov     ax, seg thunk1
                push    ax
                lea     ax, thunk1
                push    ax
                mov     ax, seg thunk2
                push    ax
                lea     ax, thunk2
                push    ax
                call    swap

; swap(A[i],i) -- 80186 & later version.

                push    seg thunk1
                push    offset thunk1
                push    seg thunk2
                push    offset thunk2
                call    swap

                .
                .
                .
; Note:         this code assumes A is an array of two byte integers.
;               Also note that we do not know which segment(s) contain
;               A and I.

thunk1          proc    far
                mov     bx, seg A   ;Need to return seg A in ES.
                push    bx          ;Save for later.
                mov     bx, seg i   ;Need segment of I in order
                mov     es, bx      ; to access it.
                mov     bx, es:i    ;Get I's value.
                shl     bx, 1
                lea     bx, A[bx]
                pop     es          ;Return segment of A[I] in es.
                ret
thunk1          endp

thunk2          proc    near
                mov     bx, seg i   ;Need to return I's seg in es.
                mov     es, bx
                lea     bx, i
                ret
thunk2          endp
```

Passing parameters by lazy evaluation is left for the programming projects.

Additional information on activation records and stack frames appears later in this chapter in the section on local variables.

11.5.10 Passing Parameters in the Code Stream

Another place where you can pass parameters is in the code stream immediately after the call instruction. The print routine in the UCR Standard Library package provides an excellent example:

```
                print
                byte    "This parameter is in the code stream.",0
```

Normally, a subroutine returns control to the first instruction immediately following the call instruction. Were that to happen here, the 80x86 would attempt to interpret the ASCII code for "This..." as an instruction. This would produce undesirable results. Fortunately, you can skip over this string when returning from the subroutine.

So how do you gain access to these parameters? Easy. The return address on the stack points at them. Consider the following implementation of print:

```
MyPrint         proc    near
                push    bp
                mov     bp, sp
                push    bx
                push    ax
                mov     bx, 2[bp]    ;Load return address into BX
PrintLp:        mov     al, cs:[bx]  ;Get next character
                cmp     al, 0        ;Check for end of string
                jz      EndStr
                putc                 ;If not end, print this char
                inc     bx           ;Move on to the next character
                jmp     PrintLp

EndStr:         inc     bx           ;Point at first byte beyond zero
                mov     2[bp], bx    ;Save as new return address
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

This procedure begins by pushing all the affected registers onto the stack. It then fetches the return address, at offset 2[BP], and prints each successive character until encountering a zero byte. Note the presence of the cs: segment override prefix in the mov al, cs:[bx] instruction. Since the data is coming from the code segment, this prefix guarantees that MyPrint fetches the character data from the proper segment. Upon encountering the zero byte, MyPrint points bx at the first byte beyond the zero. This is the address of the first instruction following the zero terminating byte. The CPU uses this value as the new return address. Now the execution of the ret instruction returns control to the instruction following the string.

The above code works great if MyPrint is a near procedure. If you need to call MyPrint from a different segment you will need to create a far procedure. Of course, the major difference is that a far return address will be on the stack at that point – you will need to use a far pointer rather than a near pointer. The following implementation of MyPrint handles this case.

```
MyPrint         proc    far
                push    bp
                mov     bp, sp
                push    bx           ;Preserve ES, AX, and BX
                push    ax
                push    es

                les     bx, 2[bp]    ;Load return address into ES:BX
PrintLp:        mov     al, es:[bx]  ;Get next character
                cmp     al, 0        ;Check for end of string
                jz      EndStr
                putc                 ;If not end, print this char
                inc     bx           ;Move on to the next character
                jmp     PrintLp

EndStr:         inc     bx           ;Point at first byte beyond zero
                mov     2[bp], bx    ;Save as new return address
                pop     es
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

Note that this code does not store es back into location [bp+4]. The reason is quite simple – es does not change during the execution of this procedure; storing es into location [bp+4] would not change the value at that location. You will notice that this version of MyPrint fetches each character from location es:[bx] rather than cs:[bx]. This is because the string you're printing is in the caller's segment, that might not be the same segment containing MyPrint.

Besides showing how to pass parameters in the code stream, the MyPrint routine also exhibits another concept: *variable length parameters*. The string following the call can be any practical length. The zero terminating byte marks the end of the parameter list. There are two easy ways to handle variable length parameters. Either use some special terminating value (like zero) or you can pass a special length value that tells the subroutine how many parameters you are passing. Both methods have their advantages and disadvantages. Using a special value to terminate a parameter list requires that you choose a value that never appears in the list. For example, MyPrint uses zero as the terminating value, so it cannot print the NULL character (whose ASCII code is zero). Sometimes this isn't a limitation. Specifying a special length parameter is another mechanism you can use to pass a variable length parameter list. While this doesn't require any special codes or limit the range of possible values that can be passed to a subroutine, setting up the length parameter and maintaining the resulting code can be a real nightmare[5].

Although passing parameters in the code stream is an ideal way to pass variable length parameter lists, you can pass fixed length parameter lists as well. The code stream is an excellent place to pass constants (like the string constants passed to MyPrint) and reference parameters. Consider the following code that expects three parameters by reference:

Calling sequence:

```
                call    AddEm
                word    I,J,K
```

Procedure:

```
AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                mov     si, [bp+2]          ;Get return address
                mov     bx, cs:[si+2]       ;Get address of J
                mov     ax, [bx]            ;Get J's value
                mov     bx, cs:[si+4]       ;Get address of K
                add     ax, [bx]            ;Add in K's value
                mov     bx, cs:[si]         ;Get address of I
                mov     [bx], ax            ;Store result
                add     si, 6               ;Skip past parms
                mov     [bp+2], si          ;Save return address
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

This subroutine adds J and K together and stores the result into I. Note that this code uses 16 bit near pointers to pass the addresses of I, J, and K to AddEm. Therefore, I, J, and K must be in the current data segment. In the example above, AddEm is a near procedure. Had it been a far procedure it would have needed to fetch a four byte pointer from the stack rather than a two byte pointer. The following is a far version of AddEm:

```
AddEm           proc    far
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                push    es
                les     si, [bp+2]          ;Get far ret adrs into es:si
                mov     bx, es:[si+2]       ;Get address of J
                mov     ax, [bx]            ;Get J's value
```

5. Especially if the parameter list changes frequently.

```
                mov     bx, es:[si+4]           ;Get address of K
                add     ax, [bx]                ;Add in K's value
                mov     bx, es:[si]             ;Get address of I
                mov     [bx], ax                ;Store result
                add     si, 6                   ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     es
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

In both versions of AddEm, the pointers to I, J, and K passed in the code stream are near pointers. Both versions assume that I, J, and K are all in the current data segment. It is possible to pass far pointers to these variables, or even near pointers to some and far pointers to others, in the code stream. The following example isn't quite so ambitious, it is a near procedure that expects far pointers, but it does show some of the major differences. For additional examples, see the exercises.

Callling sequence:

```
                call    AddEm
                dword   I,J,K
```

Code:

```
AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                push    es
                mov     si, [bp+2]              ;Get near ret adrs into si
                les     bx, cs:[si+2]           ;Get address of J into es:bx
                mov     ax, es:[bx]             ;Get J's value
                les     bx, cs:[si+4]           ;Get address of K
                add     ax, es:[bx]             ;Add in K's value
                les     bx, cs:[si]             ;Get address of I
                mov     es:[bx], ax             ;Store result
                add     si, 12                  ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     es
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Note that there are 12 bytes of parameters in the code stream this time around. This is why this code contains an add si, 12 instruction rather than the add si, 6 appearing in the other versions.

In the examples given to this point, MyPrint expects a pass by value parameter, it prints the actual characters following the call, and AddEm expects three pass by reference parameters – their addresses follow in the code stream. Of course, you can also pass parameters by value-returned, by result, by name, or by lazy evaluation in the code stream as well. The next example is a modification of AddEm that uses pass by result for I, pass by value-returned for J, and pass by name for K. This version is slightly differerent insofar as it modifies J as well as I, in order to justify the use of the value-returned parameter.

```
; AddEm(Result I:integer; ValueResult J:integer; Name K);
;
;      Computes           I:= J;
;                         J := J+K;
;
; Presumes all pointers in the code stream are near pointers.

AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si                      ;Pointer to parameter block.
                push    bx                      ;General pointer.
                push    cx                      ;Temp value for I.
                push    ax                      ;Temp value for J.

                mov     si, [bp+2]              ;Get near ret adrs into si

                mov     bx, cs:[si+2]           ;Get address of J into bx
                mov     ax, es:[bx]             ;Create local copy of J.
                mov     cx, ax                  ;Do I:=J;

                call    word ptr cs:[si+4]      ;Call thunk to get K's adrs
                add     ax, [bx]                ;Compute J := J + K

                mov     bx, cs:[si]             ;Get address of I and store
                mov     [bx], cx                ; I away.

                mov     bx, cs:[si+2]           ;Get J's address and store
                mov     [bx], ax                ; J's value away.

                add     si, 6                   ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     ax
                pop     cx
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Example calling sequences:

```
; AddEm(I,J,K)

                call    AddEm
                word    I,J,KThunk

; AddEm(I,J,A[I])

                call    AddEm
                word    I,J,AThunk
                 .
                 .
                 .
KThunk          proc    near
                lea     bx, K
                ret
KThunk          endp

AThunk          proc    near
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
AThunk          endp
```

Note: had you passed I by reference, rather than by result, in this example, the call

```
AddEm(I,J,A[i])
```

would have produced different results. Can you explain why?

Passing parameters in the code stream lets you perform some really clever tasks. The following example is considerably more complex than the others in this section, but it

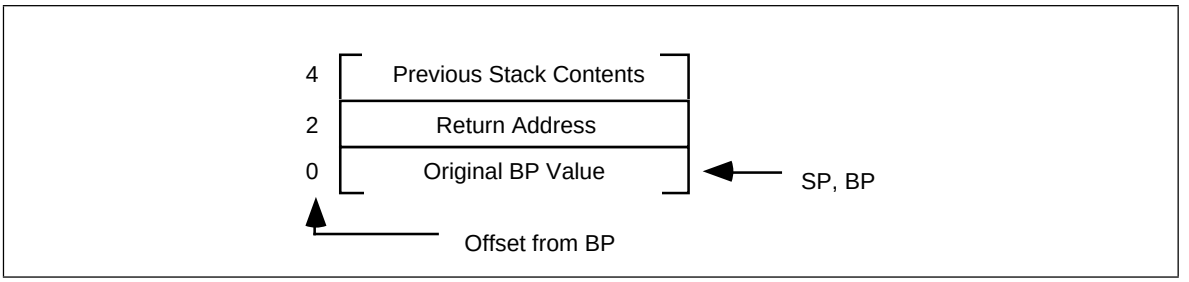


Figure 11.13 Stack Upon Entry into the ForStmt Procedure

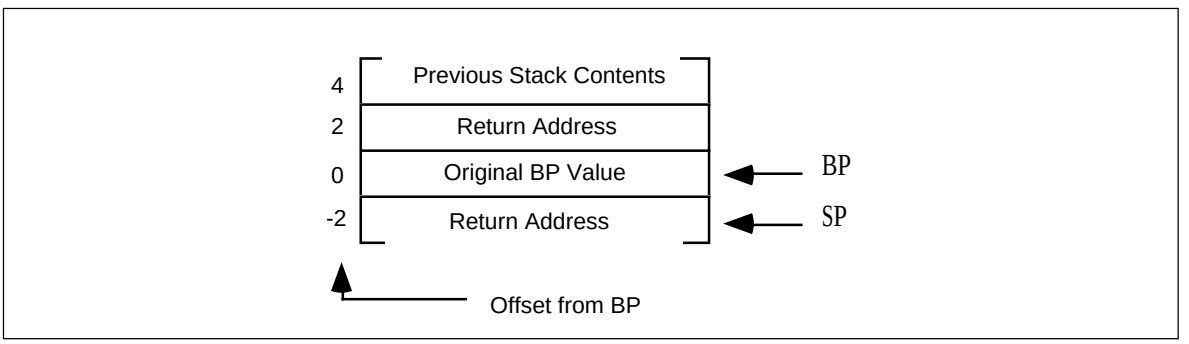


Figure 11.14 Stack Just Before Leaving the ForStmt Procedure

demonstrates the power of passing parameters in the code stream and, despite the complexity of this example, how they can simplify your programming tasks.

The following two routines implement a for/next statement, similar to that in BASIC, in assembly language. The calling sequence for these routines is the following:

```
call     ForStmt
word     «LoopControlVar», «StartValue», «EndValue»
  .
  .
« loop body statements»
  .
  .
call     Next
```

This code sets the loop control variable (whose near address you pass as the first parameter, by reference) to the starting value (passed by value as the second parameter). It then begins execution of the loop body. Upon executing the call to Next, this program would increment the loop control variable and then compare it to the ending value. If it is less than or equal to the ending value, control would return to the beginning of the loop body (the first statement following the word directive). Otherwise it would continue execution with the first statement past the call to Next.

Now you're probably wondering, "How on earth does control transfer to the beginning of the loop body?" After all, there is no label at that statement and there is no control transfer instruction instruction that jumps to the first statement after the word directive. Well, it turns out you can do this with a little tricky stack manipulation. Consider what the stack will look like upon entry into the ForStmt routine, after pushing bp onto the stack (see Figure 11.13).

Normally, the ForStmt routine would pop bp and return with a ret instruction, which removes ForStmt's activation record from the stack. Suppose, instead, ForStmt executes the following instructions:

```
add      word ptr 2[b], 2       ;Skip the parameters.
push     [bp+2]                 ;Make a copy of the rtn adrs.
mov      bp, [bp]               ;Restore bp's value.
ret                             ;Return to caller.
```

Just before the ret instruction above, the stack has the entries shown in Figure 11.14.

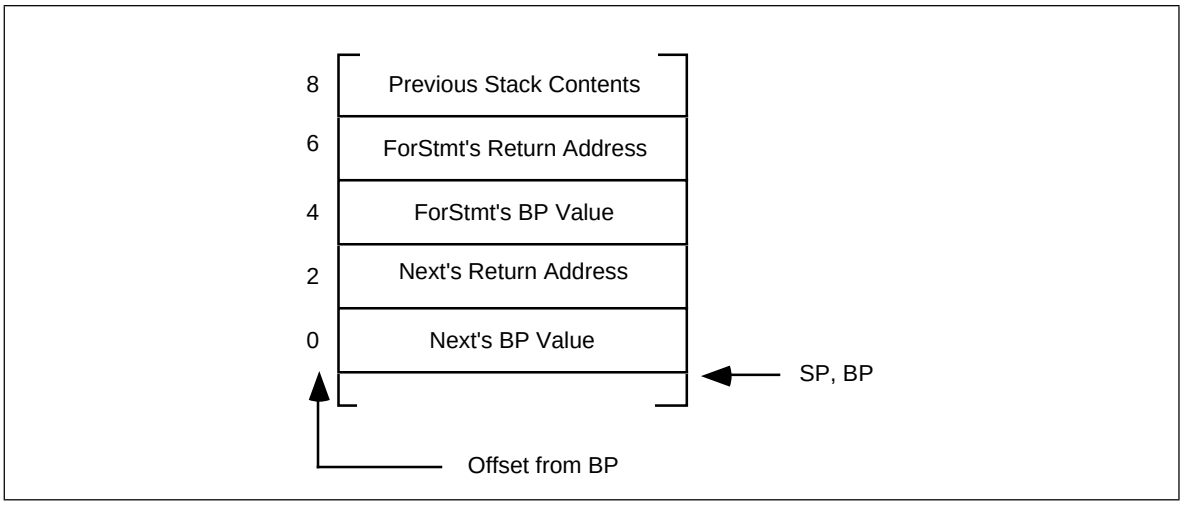


Figure 11.15 The Stack upon Entering the Next Procedure

Upon executing the ret instruction, ForStmt will return to the proper return address *but it will leave its activation record on the stack!*

After executing the statements in the loop body, the program calls the Next routine. Upon initial entry into Next (and setting up bp), the stack contains the entries appearing in Figure 11.15[6].

The important thing to see here is that ForStmt's return address, that points at the first statement past the word directive, is still on the stack and available to Next at offset [bp+6]. Next can use this return address to gain access to the parameters and return to the appropriate spot, if necessary. Next increments the loop control variable and compares it to the ending value. If the loop control variable's value is less than the ending value, Next pops its return address off the stack and returns through ForStmt's return address. If the loop control variable is greater than the ending value, Next returns through its own return address and removes ForStmt's activation record from the stack. The following is the code for Next and ForStmt:

```
                .xlist
                include  stdlib.a
                includelib stdlib.lib
                .list

dseg            segment  para public 'data'
I               word     ?
J               word     ?
dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

wp              textequ  <word ptr>

ForStmt         proc     near
                push     bp
                mov      bp, sp
                push     ax
                push     bx
                mov      bx, [bp+2]   ;Get return address
                mov      ax, cs:[bx+2];Get starting value
                mov      bx, cs:[bx]  ;Get address of var
                mov      [bx], ax     ;var := starting value
                add      wp [bp+2], 6 ;Skip over parameters
                pop      bx
```

6. Assuming the loop does not push anything onto the stack, or pop anything off the stack. Should either case occur, the ForStmt / Next loop would not work properly.

```
                pop     ax
                push    [bp+2]      ;Copy return address
                mov     bp, [bp]    ;Restore bp
                ret                 ;Leave Act Rec on stack
ForStmt         endp

Next            proc near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, [bp+6]  ;ForStmt's rtn adrs
                mov     ax, cs:[bx-2];Ending value
                mov     bx, cs:[bx-6];Ptr to loop ctrl var
                inc     wp [bx]     ;Bump up loop ctrl
                cmp     ax, [bx]    ;Is end val < loop ctrl?
                jl      QuitLoop

; If we get here, the loop control variable is less than or equal
; to the ending value. So we need to repeat the loop one more time.
; Copy ForStmt's return address over our own and then return,
; leaving ForStmt's activation record intact.

                mov     ax, [bp+6]  ;ForStmt's return address
                mov     [bp+2], ax  ;Overwrite our return address
                pop     bx
                pop     ax
                pop     bp          ;Return to start of loop body
                ret

; If we get here, the loop control variable is greater than the
; ending value, so we need to quit the loop (by returning to Next's
; return address) and remove ForStmt's activation record.

QuitLoop:       pop     bx
                pop     ax
                pop     bp
                ret     4
Next            endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                call    ForStmt
                word    I,1,5
                call    ForStmt
                word    J,2,4
                printf
                byte    "I=%d, J=%d\n",0
                dword   I,J

                call    Next        ;End of J loop
                call    Next        ;End of I loop
                print
                byte    "All Done!",cr,lf,0

Quit:           ExitPgm
Main            endp
cseg            ends
sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends
zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

The example code in the main program shows that these for loops nest exactly as you would expect in a high level language like BASIC, Pascal, or C. Of course, this is not a particularly good way to construct a for loop in assembly language. It is many times slower than using the standard loop generation techniques (see "Loops" on page 531 for more

details on that). Of course, if you don't care about speed, this is a perfectly good way to implement a loop. It is certainly easier to read and understand than the traditional methods for creating a for loop. For another (more efficient) implementation of the for loop, check out the ForLp macros in Chapter Eight (see "A Sample Macro to Implement For Loops" on page 409).

The code stream is a very convenient place to pass parameters. The UCR Standard Library makes considerable use of this parameter passing mechanism to make it easy to call certain routines. Printf is, perhaps, the most complex example, but other examples (especially in the string library) abound.

Despite the convenience, there are some disadvantages to passing parameters in the code stream. First, if you fail to provide the exact number of parameters the procedure requires, the subroutine will get very confused. Consider the UCR Standard Library print routine. It prints a string of characters up to a zero terminating byte and then returns control to the first instruction following the zero terminating byte. If you leave off the zero terminating byte, the print routine happily prints the following opcode bytes as ASCII characters until it finds a zero byte. Since zero bytes often appear in the middle of an instruction, the print routine might return control into the middle of some other instruction. This will probably crash the machine. Inserting an extra zero, which occurs more often than you might think, is another problem programmers have with the print routine. In such a case, the print routine would return upon encountering the first zero byte and attempt to execute the following ASCII characters as machine code. Once again, this usually crashes the machine.

Another problem with passing parameters in the code stream is that it takes a little longer to access such parameters. Passing parameters in the registers, in global variables, or on the stack is slightly more efficient, especially in short routines. Nevertheless, accessing parameters in the code stream isn't extremely slow, so the convenience of such parameters may outweigh the cost. Furthermore, many routines (print is a good example) are so slow anyway that a few extra microseconds won't make any difference.

11.5.11 Passing Parameters via a Parameter Block

Another way to pass parameters in memory is through a *parameter block*. A parameter block is a set of contiguous memory locations containing the parameters. To access such parameters, you would pass the subroutine a pointer to the parameter block. Consider the subroutine from the previous section that adds J and K together, storing the result in I; the code that passes these parameters through a parameter block might be

Calling sequence:

```
ParmBlock       dword   I
I               word    ?                       ;I, J, and K must appear in
J               word    ?                       ; this order.
K               word    ?
                  .
                  .
                  .
                lds     bx, ParmBlock
                call    AddEm
                  .
                  .
                  .
AddEm           proc    near
                push    ax
                mov     ax, es:2[bx]            ;Get J's value
                add     ax, es:4[bx]            ;Add in K's value
                mov     es:[bx], ax             ;Store result in I
                pop     ax
                ret
AddEm           endp
```

Note that you must allocate the three parameters in contiguous memory locations.

This form of parameter passing works well when passing several parameters by reference, because you can initialize pointers to the parameters directly within the assembler. For example, suppose you wanted to create a subroutine rotate to which you pass four parameters by reference. This routine would copy the second parameter to the first, the third to the second, the fourth to the third, and the first to the fourth. Any easy way to accomplish this in assembly is

```
; Rotate-       On entry, BX points at a parameter block in the data
;               segment that points at four far pointers. This code
;               rotates the data referenced by these pointers.

Rotate          proc    near
                push    es                      ;Need to preserve these
                push    si                      ; registers
                push    ax

                les     si, [bx+4]              ;Get ptr to 2nd var
                mov     ax, es:[si]             ;Get its value
                les     si, [bx]                ;Get ptr to 1st var
                xchg    ax, es:[si]             ;2nd->1st, 1st->ax
                les     si, [bx+12]             ;Get ptr to 4th var
                xchg    ax, es:[si]             ;1st->4th, 4th->ax
                les     si, [bx+8]              ;Get ptr to 3rd var
                xchg    ax, es:[si]             ;4th->3rd, 3rd->ax
                les     si, [bx+4]              ;Get ptr to 2nd var
                mov     es:[si], ax             ;3rd -> 2nd

                pop     ax
                pop     si
                pop     es
                ret
Rotate          endp
```

To call this routine, you pass it a pointer to a group of four far pointers in the bx register. For example, suppose you wanted to rotate the first elements of four different arrays, the second elements of those four arrays, and the third elements of those four arrays. You could do this with the following code:

```
                lea     bx, RotateGrp1
                call    Rotate
                lea     bx, RotateGrp2
                call    Rotate
                lea     bx, RotateGrp3
                call    Rotate
                  .
                  .
                  .
RotateGrp1      dword   ary1[0], ary2[0], ary3[0], ary4[0]
RotateGrp2      dword   ary1[2], ary2[2], ary3[2], ary4[2]
RotateGrp3      dword   ary1[4], ary2[4], ary3[4], ary4[4]
```

Note that the pointer to the parameter block is itself a parameter. The examples in this section pass this pointer in the registers. However, you can pass this pointer anywhere you would pass any other reference parameter – in registers, in global variables, on the stack, in the code stream, even in another parameter block! Such variations on the theme, however, will be left to your own imagination. As with any parameter, the best place to pass a pointer to a parameter block is in the registers. This text will generally adopt that policy.

Although beginning assembly language programmers rarely use parameter blocks, they certainly have their place. Some of the IBM PC BIOS and MS-DOS functions use this parameter passing mechanism. Parameter blocks, since you can initialize their values during assembly (using byte, word, etc.), provide a fast, efficient way to pass parameters to a procedure.

Of course, you can pass parameters by value, reference, value-returned, result, or by name in a parameter block. The following piece of code is a modification of the Rotate procedure above where the first parameter is passed by value (its value appears inside the parameter block), the second is passed by reference, the third by value-returned, and the

fourth by name (there is no pass by result since Rotate needs to read and write all values). For simplicity, this code uses near pointers and assumes all variables appear in the data segment:

```
; Rotate-       On entry, DI points at a parameter block in the data
;               segment that points at four pointers. The first is
;               a value parameter, the second is passed by reference,
;               the third is passed by value/return, the fourth is
;               passed by name.

Rotate          proc    near
                push    si                      ;Used to access ref parms
                push    ax                      ;Temporary
                push    bx                      ;Used by pass by name parm
                push    cx                      ;Local copy of val/ret parm

                mov     si, [di+4]              ;Get a copy of val/ret parm
                mov     cx, [si]

                mov     ax, [di]                ;Get 1st (value) parm
                call    word ptr [di+6]         ;Get ptr to 4th var
                xchg    ax, [bx]                ;1st->4th, 4th->ax
                xchg    ax, cx                  ;4th->3rd, 3rd->ax
                mov     bx, [di+2]              ;Get adrs of 2nd (ref) parm
                xchg    ax, [bx]                ;3rd->2nd, 2nd->ax
                mov     [di], ax                ;2nd->1st

                mov     bx, [di+4]              ;Get ptr to val/ret parm
                mov     [bx], cx                ;Save val/ret parm away.

                pop     cx
                pop     bx
                pop     ax
                pop     si
                ret
Rotate          endp
```

A reasonable example of a call to this routine might be:

```
I               word    10
J               word    15
K               word    20
RotateBlk       word    25, I, J, KThunk
                  .
                  .
                  .
                lea     di, RotateBlk
                call    Rotate
                  .
                  .
                  .
KThunk          proc    near
                lea     bx, K
                ret
KThunk          endp
```

11.6 Function Results

Functions return a result, which is nothing more than a result parameter. In assembly language, there are very few differences between a procedure and a function. That is probably why there aren't any "func" or "endf" directives. Functions and procedures are usually different in HLLs, function calls appear only in expressions, subroutine calls as statements[7]. Assembly language doesn't distinguish between them.

You can return function results in the same places you pass and return parameters. Typically, however, a function returns only a single value (or single data structure) as the

7. "C" is an exception to this rule. C's procedures and functions are all called functions. PL/I is another exception. In PL/I, they're all called procedures.

function result. The methods and locations used to return function results is the subject of the next three sections.

11.6.1 Returning Function Results in a Register

Like parameters, the 80x86's registers are the best place to return function results. The getc routine in the UCR Standard Library is a good example of a function that returns a value in one of the CPU's registers. It reads a character from the keyboard and returns the ASCII code for that character in the al register. Generally, functions return their results in the following registers:

Use	First	Last
Bytes:	al, ah, dl, dh, cl, ch, bl, bh	
Words:	ax, dx, cx, si, di, bx	
Double words:	dx:ax	On pre-80386
	eax, edx, ecx, esi, edi, ebx	On 80386 and later.
16-bitOffsets:	bx, si, di, dx	
32-bit Offsets	ebx, esi , edi, eax, ecx, edx	
Segmented Pointers:	es:di, es:bx, dx:ax, es:si	Do not use DS.

Once again, this table represents general guidelines. If you're so inclined, you could return a double word value in (cl, dh, al, bh). If you're returning a function result in some registers, you shouldn't save and restore those registers. Doing so would defeat the whole purpose of the function.

11.6.2 Returning Function Results on the Stack

Another good place where you can return function results is on the stack. The idea here is to push some dummy values onto the stack to create space for the function result. The function, before leaving, stores its result into this location. When the function returns to the caller, it pops everything off the stack except this function result. Many HLLs use this technique (although most HLLs on the IBM PC return function results in the registers). The following code sequences show how values can be returned on the stack:

```
function PasFunc(i,j,k:integer):integer;
begin
        PasFunc := i+j+k;
end;

m := PasFunc(2,n,l);
```

In assembly:

```
PasFunc_rtn     equ     10[bp]
PasFunc_i       equ     8[bp]
PasFunc_j       equ     6[bp]
PasFunc_k       equ     4[bp]

PasFunc         proc    near
                push    bp
                mov     bp, sp
                push    ax
                mov     ax, PasFunc_i
                add     ax, PasFunc_j
                add     ax, PasFunc_k
                mov     PasFunc_rtn, ax
                pop     ax
                pop     bp
                ret     6
PasFunc         endp
```

Calling sequence:

```
push    ax          ;Space for function return result
mov     ax, 2
push    ax
push    n
push    l
call    PasFunc
pop     ax          ;Get function return result
```

On an 80286 or later processor you could also use the code:

```
push    ax          ;Space for function return result
push    2
push    n
push    l
call    PasFunc
pop     ax          ;Get function return result
```

Although the caller pushed eight bytes of data onto the stack, PasFunc only removes six. The first "parameter" on the stack is the function result. The function must leave this value on the stack when it returns.

11.6.3 Returning Function Results in Memory Locations

Another reasonable place to return function results is in a known memory location. You can return function values in global variables or you can return a pointer (presumably in a register or a register pair) to a parameter block. This process is virtually identical to passing parameters to a procedure or function in global variables or via a parameter block.

Returning parameters via a pointer to a parameter block is an excellent way to return large data structures as function results. If a function returns an entire array, the best way to return this array is to allocate some storage, store the data into this area, and leave it up to the calling routine to deallocate the storage. Most high level languages that allow you to return large data structures as function results use this technique.

Of course, there is very little difference between returning a function result in memory and the pass by result parameter passing mechanism. See "Pass by Result" on page 576 for more details.

11.7 Side Effects

A *side effect* is any computation or operation by a procedure that isn't the primary purpose of that procedure. For example, if you elect not to preserve all affected registers within a procedure, the modification of those registers is a side effect of that procedure. Side effect programming, that is, the practice of using a procedure's side effects, is very dangerous. All too often a programmer will rely on a side effect of a procedure. Later modifications may change the side effect, invalidating all code relying on that side effect. This can make your programs hard to debug and maintain. Therefore, you should avoid side effect programming.

Perhaps some examples of side effect programming will help enlighten you to the difficulties you may encounter. The following procedure zeros out an array. For efficiency reasons, it makes the caller responsible for preserving necessary registers. As a result, one side effect of this procedure is that the bx and cx registers are modified. In particular, the cx register contains zero upon return.

```
ClrArray        proc    near
                lea     bx, array
                mov     cx, 32
ClrLoop:        mov     word ptr [bx], 0
                inc     bx
                inc     bx
                loop    ClrLoop
                ret
ClrArray        endp
```

If your code expects cx to contain zero after the execution of this subroutine, you would be relying on a side effect of the ClrArray procedure. The main purpose behind this code is zeroing out an array, not setting the cx register to zero. Later, if you modify the ClrArray procedure to the following, your code that depends upon cx containing zero would no longer work properly:

```
ClrArray        proc    near
                lea     bx, array
ClrLoop:        mov     word ptr [bx], 0
                inc     bx
                inc     bx
                cmp     bx, offset array+32
                jne     ClrLoop
                ret
ClrArray        endp
```

So how can you avoid the pitfalls of side effect programming in your procedures? By carefully structuring your code and paying close attention to exactly how your calling code and the subservient procedures interface with one another. These rules can help you avoid problems with side effect programming:

- Always properly document the input and output conditions of a procedure. Never rely on any other entry or exit conditions other than these documented operations.
- Partition your procedures so that they compute a single value or execute a single operation. Subroutines that do two or more tasks are, by definition, producing side effects unless every invocation of that subroutine requires all the computations and operations.
- When updating the code in a procedure, make sure that it still obeys the entry and exit conditions. If not, either modify the program so that it does or update the documentation for that procedure to reflect the new entry and exit conditions.
- Avoid passing information between routines in the CPU's flag register. Passing an error status in the carry flag is about as far as you should ever go. Too many instructions affect the flags and it's too easy to foul up a return sequence so that an important flag is modified on return.
- Always save and restore all registers a procedure modifies.
- Avoid passing parameters and function results in global variables.
- Avoid passing parameters by reference (with the intent of modifying them for use by the calling code).

These rules, like all other rules, were meant to be broken. Good programming practices are often sacrificed on the altar of efficiency. There is nothing wrong with breaking these rules as often as you feel necessary. However, your code will be difficult to debug and maintain if you violate these rules often. But such is the price of efficiency[8]. Until you gain enough experience to make a judicious choice about the use of side effects in your programs, you should avoid them. More often than not, the use of a side effect will cause more problems than it solves.

8. This is not just a snide remark. Expert programmers who have to wring the last bit of performance out of a section of code often resort to poor programming practices in order to achieve their goals. They are prepared, however, to deal with the problems that are often encountered in such situations and they are a lot more careful when dealing with such code.

11.8 Local Variable Storage

Sometimes a procedure will require temporary storage, that it no longer requires when the procedure returns. You can easily allocate such local variable storage on the stack.

The 80x86 supports local variable storage with the same mechanism it uses for parameters – it uses the bp and sp registers to access and allocate such variables. Consider the following Pascal program:

```
program LocalStorage;
var       i,j,k:integer;
          c: array [0..20000] of integer;

          procedure Proc1;
          var       a:array [0..30000] of integer;
                    i:integer;
          begin

          {Code that manipulates a and i}

          end;

          procedure Proc2;
          var       b:array [0..20000] of integer;
                    i:integer;
          begin

                    {Code that manipulates b and i}

          end;

begin

{main program that manipulates i,j,k, and c}

end.
```

Pascal normally allocates global variables in the data segment and local variables in the stack segment. Therefore, the program above allocates 50,002 words of local storage (30,001 words in Proc1 and 20,001 words in Proc2). This is above and beyond the other data on the stack (like return addresses). Since 50,002 words of storage consumes 100,004 bytes of storage you have a small problem – the 80x86 CPUs in real mode limit the stack segment to 65,536 bytes. Pascal avoids this problem by dynamically allocating local storage upon entering a procedure and deallocating local storage upon return. Unless Proc1 and Proc2 are both active (which can only occur if Proc1 calls Proc2 or vice versa), there is sufficient storage for this program. You don't need the 30,001 words for Proc1 and the 20,001 words for Proc2 at the same time. So Proc1 allocates and uses 60,002 bytes of storage, then deallocates this storage and returns (freeing up the 60,002 bytes). Next, Proc2 allocates 40,002 bytes of storage, uses them, deallocates them, and returns to its caller. Note that Proc1 and Proc2 share many of the same memory locations. However, they do so at different times. As long as these variables are temporaries whose values you needn't save from one invocation of the procedure to another, this form of local storage allocation works great.

The following comparison between a Pascal procedure and its corresponding assembly language code will give you a good idea of how to allocate local storage on the stack:

```
procedure LocalStuff(i,j,k:integer);
var l,m,n:integer; {local variables}
begin

       l := i+2;
       j := l*k+j;
       n := j-l;
       m := l+j+n;

end;
```

Calling sequence:

```
        LocalStuff(1,2,3);
```

Assembly language code:

```
LStuff_i        equ     8[bp]
LStuff_j        equ     6[bp]
LStuff_k        equ     4[bp]
LStuff_l        equ     -4[bp]
LStuff_m        equ     -6[bp]
LStuff_n        equ     -8[bp]

LocalStuff      proc    near
                push    bp
                mov     bp, sp
                push    ax
                sub     sp, 6                   ;Allocate local variables.
L0:             mov     ax, LStuff_i
                add     ax, 2
                mov     LStuff_l, ax
                mov     ax, LStuff_l
                mul     LStuff_k
                add     ax, LStuff_j
                mov     LStuff_j, ax
                sub     ax, LStuff_l            ;AX already contains j
                mov     LStuff_n, ax
                add     ax, LStuff_l            ;AX already contains n
                add     ax, LStuff_j
                mov     LStuff_m, ax

                add     sp, 6                   ;Deallocate local storage
                pop     ax
                pop     bp
                ret     6
LocalStuff      endp
```

The `sub sp, 6` instruction makes room for three words on the stack. You can allocate `l`, `m`, and `n` in these three words. You can reference these variables by indexing off the `bp` register using negative offsets (see the code above). Upon reaching the statement at label `L0`, the stack looks something like Figure 11.15.

This code uses the matching `add sp, 6` instruction at the end of the procedure to deallocate the local storage. The value you add to the stack pointer must exactly match the value you subtract when allocating this storage. If these two values don't match, the stack pointer upon entry to the routine will not match the stack pointer upon exit; this is like pushing or popping too many items inside the procedure.

Unlike parameters, that have a fixed offset in the activation record, you can allocate local variables in any order. As long as you are consistent with your location assignments, you can allocate them in any way you choose. Keep in mind, however, that the 80x86 supports two forms of the `disp[bp]` addressing mode. It uses a one byte displacement when it is in the range -128..+127. It uses a two byte displacement for values in the range -32,768..+32,767. Therefore, you should place all primitive data types and other small structures close to the base pointer, so you can use single byte displacements. You should place large arrays and other data structures below the smaller variables on the stack.

Most of the time you don't need to worry about allocating local variables on the stack. Most programs don't require more than 64K of storage. The CPU processes global variables faster than local variables. There are two situations where allocating local variables as globals in the data segment is not practical: when interfacing assembly language to HLLs like Pascal, and when writing recursive code. When interfacing to Pascal, your assembly language code may not have a data segment it can use, recursion often requires multiple instances of the same local variable.

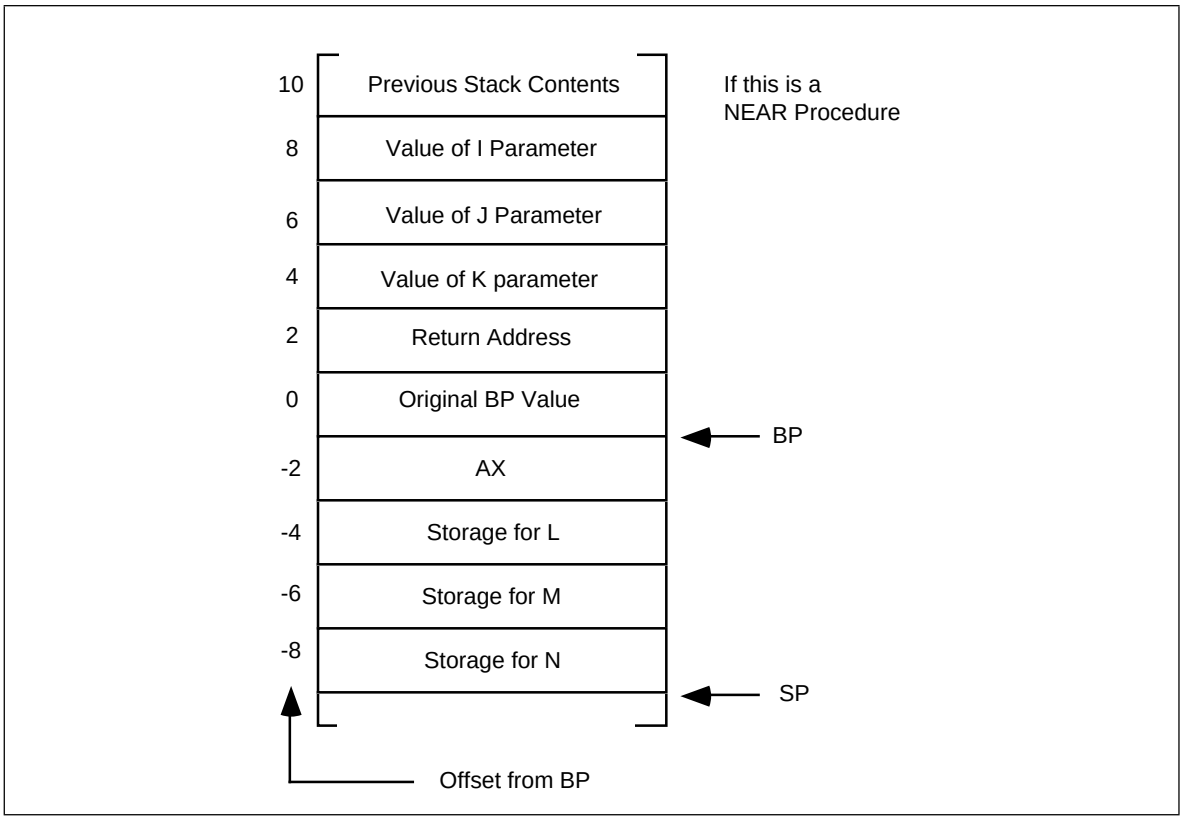


Figure 11.16 The Stack upon Entering the Next Procedure

11.9 Recursion

Recursion occurs when a procedure calls itself. The following, for example, is a recursive procedure:

```
Recursive       proc
                call    Recursive
                ret
Recursive       endp
```

Of course, the CPU will never execute the ret instruction at the end of this procedure. Upon entry into Recursive, this procedure will immediately call itself again and control will never pass to the ret instruction. In this particular case, run away recursion results in an infinite loop.

In many respects, recursion is very similar to iteration (that is, the repetitive execution of a loop). The following code also produces an infinite loop:

```
Recursive       proc
                jmp     Recursive
                ret
Recursive       endp
```

There is, however, one major difference between these two implementations. The former version of Recursive pushes a return address onto the stack with each invocation of the subroutine. This does not happen in the example immediately above (since the jmp instruction does not affect the stack).

Like a looping structure, recursion requires a termination condition in order to stop infinite recursion. Recursive could be rewritten with a termination condition as follows:

```
Recursive       proc
                dec     ax
                jz      QuitRecurse
                call    Recursive
QuitRecurse:    ret
Recursive       endp
```

This modification to the routine causes Recursive to call itself the number of times appearing in the ax register. On each call, Recursive decrements the ax register by one and calls itself again. Eventually, Recursive decrements ax to zero and returns. Once this happens, the CPU executes a string of ret instructions until control returns to the original call to Recursive.

So far, however, there hasn't been a real need for recursion. After all, you could efficiently code this procedure as follows:

```
Recursive       proc
RepeatAgain:    dec     ax
                jnz     RepeatAgain
                ret
Recursive       endp
```

Both examples would repeat the body of the procedure the number of times passed in the ax register[9]. As it turns out, there are only a few recursive algorithms that you cannot implement in an iterative fashion. However, many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

The quicksort algorithm is probably the most famous algorithm that almost always appears in recursive form. A Pascal implementation of this algorithm follows:

```
procedure quicksort(var a:ArrayToSort; Low,High: integer);

        procedure sort(l,r: integer);
        var i,j,Middle,Temp: integer;
        begin

                i:=l;
                j:=r;
                Middle:=a[(l+r) DIV 2];
                repeat
                        while (a[i] < Middle) do i:=i+1;
                        while (Middle < a[j]) do j:=j-1;
                        if (i <= j) then begin

                          Temp:=a[i];
                          a[i]:=a[j];
                          a[j]:=Temp;
                          i:=i+1;
                          j:=j-1;

                        end;

                until i>j;
                if l<j then sort(l,j);
                if i<r then sort(i,r);

        end;

begin {quicksort};

        sort(Low,High);

end;
```

The sort subroutine is the recursive routine in this package. Recursion occurs at the last two if statements in the sort procedure.

In assembly language, the sort routine looks something like this:

9. Although the latter version will do it considerably faster since it doesn't have the overhead of the CALL/RET instructions.

```
                include     stdlib.a
                includelib  stdlib.lib
cseg            segment
                assume      cs:cseg, ds:cseg, ss:sseg, es:cseg

; Main program to test sorting routine

Main            proc
                mov     ax, cs
                mov     ds, ax
                mov     es, ax

                mov     ax, 0
                push    ax
                mov     ax, 31
                push    ax
                call    sort

                ExitPgm                         ;Return to DOS
Main            endp

; Data to be sorted

a               word    31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16
                word    15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

; procedure sort (l,r:integer)
; Sorts array A between indices l and r

l               equ     6[bp]
r               equ     4[bp]
i               equ     -2[bp]
j               equ     -4[bp]

sort            proc    near
                push    bp
                mov     bp, sp
                sub     sp, 4                   ;Make room for i and j.

                mov     ax, l                   ;i := l
                mov     i, ax
                mov     bx, r                   ;j := r

                mov     j, bx

; Note: This computation of the address of a[(l+r) div 2] is kind
; of strange. Rather than divide by two, then multiply by two
; (since A is a word array), this code simply clears the L.O. bit
; of BX.

                add     bx, l                   ;Middle := a[(l+r) div 2]
                and     bx, 0FFFEh
                mov     ax, a[bx]               ;BX*2, because this is a word
;                                               ; array,nullifies the “div 2”
;                                               ; above.
;
; Repeat until i > j: Of course, I and J are in BX and SI.

                lea     bx, a                   ;Compute the address of a[i]
                add     bx, i                   ; and leave it in BX.
                add     bx, i

                lea     si, a                   ;Compute the address of a[j]
                add     si, j                   ; and leave it in SI.
                add     si, j

RptLp:

; While (a [i] < Middle) do i := i + 1;

                sub     bx, 2                   ;We’ll increment it real
soon.
WhlLp1:         add     bx, 2
                cmp     ax, [bx]                ;AX still contains middle
                jg      WhlLp1

; While (Middle < a[j]) do j := j-1
```

```
                add     si, 2                   ;We'll decrement it in loop
WhlLp2:         add     si, 2
                cmp     ax, [si]                ;AX still contains middle
                jl      WhlLp2                  ; value.
                cmp     bx, si
                jnle    SkipIf

; Swap, if necessary

                mov     dx, [bx]
                xchg    dx, [si]
                xchg    dx, [bx]

                add     bx, 2                   ;Bump by two (integer values)
                sub     si, 2

SkipIf:         cmp     bx, si
                jng     RptLp

; Convert SI and BX back to I and J

                lea     ax, a
                sub     bx, ax
                shr     bx, 1
                sub     si, ax
                        shrsi, 1

; Now for the recursive part:

                mov     ax, l
                cmp     ax, si
                jnl     NoRec1
                push    ax
                push    si
                call    sort

NoRec1:         cmp     bx, r
                jnl     NoRec2
                push    bx
                push    r
                call    sort
NoRec2:         mov     sp, bp
                pop     bp
                ret     4

Sort            endp

cseg            ends
sseg            segment stack 'stack'
                word    256 dup (?)
sseg            ends
                end     main
```

Other than some basic optimizations (like keeping several variables in registers), this code is almost a literal translation of the Pascal code. Note that the local variables i and j aren't necessary in this assembly language code (we could use registers to hold their values). Their use simply demonstrates the allocation of local variables on the stack.

There is one thing you should keep in mind when using recursion – recursive routines can eat up a considerable stack space. Therefore, when writing recursive subroutines, always allocate sufficient memory in your stack segment. The example above has an extremely anemic 512 byte stack space, however, it only sorts 32 numbers therefore a 512 byte stack is sufficient. In general, you won't know the depth to which recursion will take you, so allocating a large block of memory for the stack may be appropriate.

There are several efficiency considerations that apply to recursive procedures. For example, the second (recursive) call to sort in the assembly language code above need not be a recursive call. By setting up a couple of variables and registers, a simple jmp instruction can can replace the pushes and the recursive call. This will improve the performance of the quicksort routine (quite a bit, actually) and will reduce the amount of memory the stack requires. A good book on algorithms, such as D.E. Knuth's The Art of Computer Programming, Volume 3, would be an excellent source of additional material on quick-

sort. Other texts on algorithm complexity, recursion theory, and algorithms would be a good place to look for ideas on efficiently implementing recursive algorithms.

11.10 Sample Program

The following sample program demonstrates several concepts appearing in this chapter, most notably, passing parameters on the stack. This program (Pgm11_1.asm appearing on the companion CD-ROM) manipulates the PC's memory-mapped text video display screen (at address B800:0 for color displays, B000:0 for monochrome displays). It provides routines that "capture" all the data on the screen to an array, write the contents of an array to the screen, clear the screen, scroll one line up or down, position the cursor at an (X,Y) coordinate, and retrieve the current cursor position.

Note that this code was written to demonstrate the use of parameters and local variables. Therefore, it is rather inefficient. As the comments point out, many of the functions this package provides could be written to run much faster using the 80x86 string instructions. See the laboratory exercises for a different version of some of these functions that is written in such a fashion. Also note that this code makes some calls to the PC's BIOS to set and obtain the cursor position as well as clear the screen. See the chapter on BIOS and DOS for more details on these BIOS calls.

```
; Pgm11_1.asm
;
; Screen Aids.
;
; This program provides some useful screen manipulation routines
; that let you do things like position the cursor, save and restore
; the contents of the display screen, clear the screen, etc.
;
; This program is not very efficient.  It was written to demonstrate
; parameter passing, use of local variables, and direct conversion of
; loops to assembly language.  There are far better ways of doing
; what this program does (running about 5-10x faster) using the 80x86
; string instructions.



                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

                .386                    ;Comment out these two statements
                option    segment:use16 ; if you are not using an 80386.


; ScrSeg- This is the video screen's segment address.  It should be
;                B000 for mono screens and B800 for color screens.

ScrSeg          =         0B800h

dseg            segment   para public 'data'

XPosn           word      ?             ;Cursor X-Coordinate (0..79)
YPosn           word      ?             ;Cursor Y-Coordinate (0..24)

; The following array holds a copy of the initial screen data.

SaveScr         word      25 dup (80 dup (?))

dseg            ends


cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg
```

```
; Capture-       Copies the data on the screen to the array passed
;                by reference as a parameter.
;
; procedure Capture(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;      for y := 0 to 24 do
;          for x := 0 to 79 do
;                SCREEN[y,x] := ScrCopy[y,x];
; end;
;
;
; Activation record for Capture:
;
;      |                       |
;      | Previous stk contents |
;      -------------------------
;      |  ScrCopy Seg Adrs     |
;      --                     --
;      | ScrCopy offset Adrs   |
;      -------------------------
;      | Return Adrs (near)    |
;      -------------------------
;      |      Old BP           |
;      ------------------------- <- BP
;      |  X coordinate value   |
;      -------------------------
;      |  Y coordinate value   |
;      -------------------------
;      | Registers, etc.       |
;      ------------------------- <- SP


ScrCopy_cap     textequ  <dword ptr [bp+4]>
X_cap           textequ  <word ptr [bp-2]>
Y_cap           textequ  <word ptr [bp-4]>

Capture         proc
                push     bp
                mov      bp, sp
                sub      sp, 4        ;Allocate room for locals.

                push     es
                push     ds
                push     ax
                push     bx
                push     di

                mov      bx, ScrSeg   ;Set up pointer to SCREEN
                mov      es, bx       ; memory (ScrSeg:0).

                lds      di, ScrCopy_cap ;Get ptr to capture array.

                mov      Y_cap, 0
YLoop:          mov      X_cap, 0
XLoop:          mov      bx, Y_cap
                imul     bx, 80       ;Screen memory is a 25x80 array
                add      bx, X_cap    ; stored in row major order
                add      bx, bx       ; with two bytes per element.

                mov      ax, es:[bx]  ;Read character code from screen.
                mov      [di][bx], ax ;Store away into capture array.

                inc      X_Cap        ;Repeat for each character on this
                cmp      X_Cap, 80    ; row of characters (each character
                jb       XLoop        ; in the row is two bytes).

                inc      Y_Cap        ;Repeat for each row on the screen.
```

```
                cmp     Y_Cap, 25
                jb      YLoop

                pop     di
                pop     bx
                pop     ax
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret     4
Capture         endp


; Fill-         Copies array passed by reference onto the screen.
;
; procedure Fill(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;     for y := 0 to 24 do
;         for x := 0 to 79 do
;               ScrCopy[y,x] := SCREEN[y,x];
; end;
;
;
; Activation record for Fill:
;
;     |                       |
;     | Previous stk contents |
;     -------------------------
;     |  ScrCopy Seg Adrs     |
;     --                     --
;     | ScrCopy offset Adrs   |
;     -------------------------
;     | Return Adrs (near)    |
;     -------------------------
;     |      Old BP           |
;     ------------------------- <- BP
;     |  X coordinate value   |
;     -------------------------
;     |  Y coordinate value   |
;     -------------------------
;     | Registers, etc.       |
;     ------------------------- <- SP


ScrCopy_fill    texequ  <dword ptr [bp+4]>
X_fill          texequ  <word ptr [bp-2]>
Y_fill          texequ  <word ptr [bp-4]>

Fill            proc
                push    bp
                mov     bp, sp
                sub     sp, 4

                push    es
                push    ds
                push    ax
                push    bx
                push    di

                mov     bx, ScrSeg   ;Set up pointer to SCREEN
                mov     es, bx       ; memory (ScrSeg:0).

                lds     di, ScrCopy_fill ;Get ptr to data array.

                mov     Y_Fill, 0
YLoop:          mov     X_Fill, 0
```

```
XLoop:          mov     bx, Y_Fill
                imul    bx, 80      ;Screen memory is a 25x80 array
                add     bx, X_Fill  ; stored in row major order
                add     bx, bx      ; with two bytes per element.

                mov     ax, [di][bx] ;Store away into capture array.
                mov     es:[bx], ax  ;Read character code from screen.

                inc     X_Fill      ;Repeat for each character on this
                cmp     X_Fill, 80  ; row of characters (each character
                jb      XLoop       ; in the row is two bytes).

                inc     Y_Fill      ;Repeat for each row on the screen.
                cmp     Y_Fill, 25
                jb      YLoop

                pop     di
                pop     bx
                pop     ax
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret     4
Fill            endp


; Scroll_up-    Scrolls the screen up on line.  It does this by copying the
;               second line to the first, the third line to the second, the
;               fourth line to the third, etc.
;
; procedure Scroll_up;
; var x,y:integer;
; begin
;       for y := 1 to 24 do
;           for x := 0 to 79 do
;                 SCREEN[Y-1,X] := SCREEN[Y,X];
; end;
;
; Activation record for Scroll_up:
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       | Return Adrs (near)    |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       | Registers, etc.       |
;       ------------------------- <- SP


X_su            textequ  <word ptr [bp-2]>
Y_su            textequ  <word ptr [bp-4]>

Scroll_up       proc
                push    bp
                mov     bp, sp
                sub     sp, 4       ;Make room for X, Y.

                push    ds
                push    ax
                push    bx

                mov     ax, ScrSeg
                mov     ds, ax
```

```
                mov     Y_su, 0
su_Loop1:       mov     X_su, 0

su_Loop2:       mov     bx, Y_su    ;Compute index into screen
                imul    bx, 80      ; array.
                add     bx, X_su
                add     bx, bx      ;Remember, this is a word array.

                mov     ax, [bx+160] ;Fetch word from source line.
                mov     [bx], ax    ;Store into dest line.

                inc     X_su
                cmp     X_su, 80
                jb      su_Loop2

                inc     Y_su
                cmp     Y_su, 80
                jb      su_Loop1

                pop     bx
                pop     ax
                pop     ds
                mov     sp, bp
                pop     bp
                ret
Scroll_up       endp


; Scroll_dn-    Scrolls the screen down one line.  It does this by copying the
;               24th line to the 25th, the 23rd line to the 24th, the 22nd line
;               to the 23rd, etc.
;
; procedure Scroll_dn;
; var x,y:integer;
; begin
;     for y := 23 downto 0 do
;         for x := 0 to 79 do
;               SCREEN[Y+1,X] := SCREEN[Y,X];
; end;
;
; Activation record for Scroll_dn:
;
;     |                       |
;     | Previous stk contents |
;     -------------------------
;     | Return Adrs (near)    |
;     -------------------------
;     |      Old BP           |
;     ------------------------- <- BP
;     |  X coordinate value   |
;     -------------------------
;     |  Y coordinate value   |
;     -------------------------
;     | Registers, etc.       |
;     ------------------------- <- SP


X_sd            textequ  <word ptr [bp-2]>
Y_sd            textequ  <word ptr [bp-4]>

Scroll_dn       proc
                push    bp
                mov     bp, sp
                sub     sp, 4       ;Make room for X, Y.

                push    ds
                push    ax
                push    bx

                mov     ax, ScrSeg
```

```
                mov     ds, ax
                mov     Y_sd, 23
sd_Loop1:       mov     X_sd, 0

sd_Loop2:       mov     bx, Y_sd    ;Compute index into screen
                imul    bx, 80      ; array.
                add     bx, X_sd
                add     bx, bx      ;Remember, this is a word array.

                mov     ax, [bx]    ;Fetch word from source line.
                mov     [bx+160], ax ;Store into dest line.

                inc     X_sd
                cmp     X_sd, 80
                jb      sd_Loop2

                dec     Y_sd
                cmp     Y_sd, 0
                jge     sd_Loop1

                pop     bx
                pop     ax
                pop     ds
                mov     sp, bp
                pop     bp
                ret
Scroll_dn       endp



; GotoXY-       Positions the cursor at the specified X, Y coordinate.
;
; procedure gotoxy(x,y:integer);
; begin
;       BIOS(posnCursor,x,y);
; end;
;
; Activation record for GotoXY
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       | Return Adrs (near)    |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


X_gxy           textequ  <byte ptr [bp+6]>
Y_gxy           textequ  <byte ptr [bp+4]>

GotoXY          proc
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    dx

                mov     ah, 2       ;Magic BIOS value for gotoxy.
                mov     bh, 0       ;Display page zero.
                mov     dh, Y_gxy   ;Set up BIOS (X,Y) parameters.
                mov     dl, X_gxy
                int     10h         ;Make the BIOS call.
```

```
                pop     dx
                pop     bx
                pop     ax
                mov     sp, bp
                pop     bp
                ret     4
GotoXY          endp


; GetX-         Returns cursor X-Coordinate in the AX register.

GetX            proc
                push    bx
                push    cx
                push    dx

                mov     ah, 3       ;Read X, Y coordinates from
                mov     bh, 0       ; BIOS
                int     10h

                mov     al, dl      ;Return X coordinate in AX.
                mov     ah, 0

                pop     dx
                pop     cx
                pop     bx
                ret
GetX            endp



; GetY-         Returns cursor Y-Coordinate in the AX register.

GetY            proc
                push    bx
                push    cx
                push    dx

                mov     ah, 3
                mov     bh, 0
                int     10h

                mov     al, dh      ;Return Y Coordinate in AX.
                mov     ah, 0

                pop     dx
                pop     cx
                pop     bx
                ret
GetY            endp



; ClearScrn-    Clears the screen and positions the cursor at (0,0).
;
; procedure ClearScrn;
; begin
;       BIOS(Initialize)
; end;

ClearScrn       proc
                push    ax
                push    bx
                push    cx
                push    dx

                mov     ah, 6       ;Magic BIOS number.
                mov     al, 0       ;Clear entire screen.
                mov     bh, 07      ;Clear with black spaces.
```

```
                mov     cx, 0000;Upper left corner is (0,0)
                mov     dl, 79      ;Lower X-coordinate
                mov     dh, 24      ;Lower Y-coordinate
                int     10h         ;Make the BIOS call.

                push    0           ;Position the cursor to (0,0)
                push    0           ; after the call.
                call    GotoXY

                pop     dx
                pop     cx
                pop     bx
                pop     ax
                ret
ClearScrn       endp


; A short main program to test out the above:

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; Save the screen as it looks when this program is run.

                push    seg SaveScr
                push    offset SaveScr
                call    Capture

                call    GetX
                mov     XPosn, ax

                call    GetY
                mov     YPosn, ax


; Clear the screen to prepare for our stuff.

                call    ClearScrn

; Position the cursor in the middle of the screen and print some stuff.

                push    30          ;X value
                push    10          ;Y value
                call    GotoXY

                print
                byte    "Screen Manipulatation Demo",0

                push    30
                push    11
                call    GotoXY

                print
                byte    "Press any key to continue",0

                getc


; Scroll the screen up two lines

                call    Scroll_up
                call    Scroll_up
                getc

;Scroll the screen down four lines:

                call    Scroll_dn
```

```
                call    Scroll_dn
                call    Scroll_dn
                call    Scroll_dn
                getc

; Restore the screen to what it looked like prior to this call.

                push    seg SaveScr
                push    offset SaveScr
                call    Fill

                push    XPosn
                push    YPosn
                call    GotoXY

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

11.11 Laboratory Exercises

This laboratory exercise demonstrates how a C/C++ program calls some assembly language functions. This exercise consists of two program units: a Borland C++ program (Ex11_1.cpp) and a MASM 6.11 program (Ex11_1a.asm). Since you may not have access to a C++ compiler (and Borland C++ in particular)[10], the EX11.EXE file contains a precompiled and linked version of these files. If you have a copy of Borland C++ then you can compile/assemble these files using the makefile that also appears in the Chapter 11 subdirectory.

The C++ program listing appears in Section 11.11.1. This program clears the screen and then bounces a pound sign ("#") around the screen until the user presses any key. Then this program restores the screen to the previous display before running the program and quits. All screen manipulation, as well as testing for a keypress, is taken care of by functions written in assembly language. The "extern" statements at the beginning of the program provide the linkage to these assembly language functions[11]. There are a few important things to note about how C/C++ passes parameters to an assembly language function:

- C++ pushes parameters on the stack in the *reverse* order that they appear in a parameter list. For example, for the call "f(a,b);" C++ would push b first and a second. This is opposite of most of the examples in this chapter.
- In C++, the caller is responsible for removing parameters from the stack. In this chapter, the callee (the function itself) usually removed the parameters by specifying some value after the ret instruction. Assembly language functions that C++ calls must *not* do this.
- C++ on the PC uses different memory models to control whether pointers and functions are near or far. This particular program uses the *compact*

10. There is nothing Borland specific in this C++ program. Borland was chosen because it provides an option that generates well annotated assembly output.

11. The *extern "C"* phrase instructs Borland C++ to generate standard C external names rather than C++ *mangled* names. A C external name is the function name with an underscore in front of it (e.g., GotoXY becomes _GotoXY). C++ completely changes the name to handle overloading and it is difficult to predict the actual name of the corresponding assembly language function.

memory model. This provides for near procedures and far pointers. Therefore, all calls will be near (with only a two-byte return address on the stack) and all pointers to data objects will be far.

- Borland C++ requires a function to preserve the segment registers, BP, DI, and SI. The function need not preserve any other registers. If an assembly language function needs to return a 16-bit function result to C++, it must return this value in the AX register.
- See the Borland C++ Programmer's Guide (or corresponding manual for your C++ compiler) for more details about the C/C++ and assembly language interface.

Most C++ compilers give you the option of generating assembly language output rather than binary machine code. Borland C++ is nice because it generates nicely annotated assembly output with comments pointing out which C++ statments correspond to a given sequence of assembly language instructions. The assembly language output of BCC appears in Section 11.11.2 (This is a slightly edited version to remove some superfluous information). Look over this code and note that, subject to the rules above, the C++ compiler emits code that is very similar to that described throughout this chapter.

The Ex11_1a.asm file (see section 11.11.3) is the actual assembly code the C++ program calls. This contains the functions for the GotoXY, GetXY, ClrScrn, tstKbd, Capture, PutScrn, PutChar, and PutStr routines that Ex11_1.cpp calls. To avoid legal software distribution problems, this particular C/C++ program does not include any calls to C/C++ Standard Library functions. Furthermore, it does not use the standard C0m.obj file from Borland that calls the main program. Borland's liberal license agreement does *not* allow one to distribute their librarys and object modules unlinked with other code. The assembly language code provides the necessary I/O routines and it also provides a startup routine (StartPgm) that call the C++ main program when DOS/Windows transfers control to the program. By supplying the routines this way, you do not need the Borland libraries or object code to link these programs together.

One side effect of linking the modules in this fashion is that the compiler, assembler, and linker cannot store the correct source level debugging information in the .exe file. Therefore, you will not be able to use CodeView to view the actual source code. Instead, you will have to work with disassembled machine code. This is where the assembly output from Borland C++ (Ex11_1.asm) comes in handy. As you single step through the main C++ program you can trace the program flow by looking at the Ex11_1.asm file.

For your lab report: single step through the StartPgm code until it calls the C++ main function. When this happens, locate the calls to the routines in Ex11_1a.asm. Set breakpoints on each of these calls using the F9 key. Run up to each breakpoint and then single step into the function using the F8 key. Once inside, display the memory locations starting at SS:SP. Identify each parameter passed on the stack. For reference parameters, you may want to look at the memory locations whose address appears on the stack. Report your findings in your lab report.

Include a printout of the Ex11_1.asm file and identify those instructions that push each parameter onto the stack. At run time, determine the values that each parameter push sequence pushes onto the stack and include these values in your lab report.

Many of the functions in the assembly file take a considerable amount of time to execute. Therefore, you should not single step through each of the functions. Instead, make sure you've set up the breakpoints on each of the call instructions in the C++ program and use the F5 key to run (at full speed) up to the next function call.

11.11.1 Ex11_1.cpp

```
extern "C" void GotoXY(unsigned y, unsigned x);
extern "C" void GetXY(unsigned &x, unsigned &y);
extern "C" void ClrScrn();
extern "C" int tstKbd();
```

```
extern "C" void Capture(unsigned ScrCopy[25][80]);
extern "C" void PutScr(unsigned ScrCopy[25][80]);
extern "C" void PutChar(char ch);
extern "C" void PutStr(char *ch);

int main()
{
    unsigned SaveScr[25][80];

    int         dx,
                x,
                dy,
                y;

    long        i;

    unsigned    savex,
                savey;



    GetXY(savex, savey);
    Capture(SaveScr);
    ClrScrn();

    GotoXY(24,0);
    PutStr("Press any key to quit");

    dx = 1;
    dy = 1;
    x = 1;
    y = 1;
    while (!tstKbd())
    {

       GotoXY(y, x);
        PutChar('#');

        for (i=0; i<500000; ++i);

        GotoXY(y, x);
        PutChar(' ');

       x += dx;
        y += dy;
        if (x >= 79)
       {
            x = 78;
            dx = -1;
       }
        else if (x <= 0)
       {
            x = 1;
            dx = 1;
       }

        if (y >= 24)
       {
            y = 23;
            dy = -1;
       }
        else if (y <= 0)
       {
            y = 1;
            dy = 1;
       }

    }

    PutScr(SaveScr);
```

```
    GotoXY(savey, savex);
    return 0;
}
```

11.11.2 Ex11_1.asm

```
_TEXT  segment byte public 'CODE'
_TEXT  ends
DGROUP group     _DATA,_BSS
       assume    cs:_TEXT,ds:DGROUP
_DATA  segment word public 'DATA'
d@     label     byte
d@w    label     word
_DATA  ends
_BSS   segment word public 'BSS'
b@     label     byte
b@w    label     word
_BSS   ends

_TEXT  segment byte public 'CODE'
   ;
   ;   int main()
   ;
       assume    cs:_TEXT
_main  proc      near
       push      bp
       mov       bp,sp
       sub       sp,4012
       push      si
       push      di
   ;
   ;   {
   ;       unsigned SaveScr[25][80];
   ;
   ;       int         dx,
   ;                   x,
   ;                   dy,
   ;                   y;
   ;
   ;       long        i;
   ;
   ;       unsigned    savex,
   ;                   savey;
   ;
   ;
   ;
   ;       GetXY(savex, savey);
   ;
       push      ss
       lea       ax,word ptr [bp-12]
       push      ax
       push      ss
       lea       ax,word ptr [bp-10]
       push      ax
       call      near ptr _GetXY
       add       sp,8
   ;
   ;       Capture(SaveScr);
   ;
       push      ss
       lea       ax,word ptr [bp-4012]
       push      ax
       call      near ptr _Capture
       pop       cx
       pop       cx
   ;
```

```
;       ClrScrn();
;
        call    near ptr _ClrScrn
;
;
;       GotoXY(24,0);
;
        xor     ax,ax
        push    ax
        mov     ax,24
        push    ax
        call    near ptr _GotoXY
        pop     cx
        pop     cx
;
;       PutStr("Press any key to quit");
;
        push    ds
        mov     ax,offset DGROUP:s@
        push    ax
        call    near ptr _PutStr
        pop     cx
        pop     cx
;
;
;       dx = 1;
;
        mov     word ptr [bp-2],1
;
;       dy = 1;
;
        mov     word ptr [bp-4],1
;
;       x = 1;
;
        mov     si,1
;
;       y = 1;
;
        mov     di,1
        jmp     @1@422
@1@58:
;
;       while (!tstKbd())
;       {
;
;             GotoXY(y, x);
;
        push    si
        push    di
        call    near ptr _GotoXY
        pop     cx
        pop     cx
;
;           PutChar('#');
;
        mov     al,35
        push    ax
        call    near ptr _PutChar
        pop     cx
;
;
;           for (i=0; i<500000; ++i);
;
        mov     word ptr [bp-6],0
        mov     word ptr [bp-8],0
        jmp     short @1@114
@1@86:
        add     word ptr [bp-8],1
        adc     word ptr [bp-6],0
```

```
@1@114:
       cmp       word ptr [bp-6],7
       jl        short @1@86
       jne       short @1@198
       cmp       word ptr [bp-8],-24288
       jb        short @1@86
@1@198:
   ;
   ;
   ;           GotoXY(y, x);
   ;
       push      si
       push      di
       call      near ptr _GotoXY
       pop       cx
       pop       cx
   ;
   ;           PutChar(' ');
   ;
       mov       al,32
       push      ax
       call      near ptr _PutChar
       pop       cx
   ;
   ;
   ;
   ;
   ;             x += dx;
   ;
       add       si,word ptr [bp-2]
   ;
   ;           y += dy;
   ;
       add       di,word ptr [bp-4]
   ;
   ;           if (x >= 79)
   ;
       cmp       si,79
       jl        short @1@254
   ;
   ;             {
   ;               x = 78;
   ;
       mov       si,78
   ;
   ;               dx = -1;
   ;
       mov       word ptr [bp-2],-1
   ;
   ;             }
   ;
       jmp       short @1@310
@1@254:
   ;
   ;           else if (x <= 0)
   ;
       or        si,si
       jg        short @1@310
   ;
   ;             {
   ;               x = 1;
   ;
       mov       si,1
   ;
   ;               dx = 1;
   ;
       mov       word ptr [bp-2],1
@1@310:
   ;
   ;             }
```

```
    ;
    ;           if (y >= 24)
    ;
        cmp       di,24
        jl        short @1@366
    ;
    ;             {
    ;               y = 23;
    ;
        mov       di,23
    ;
    ;               dy = -1;
    ;
        mov       word ptr [bp-4],-1
    ;
    ;             }
    ;
        jmp       short @1@422
@1@366:
    ;
    ;           else if (y <= 0)
    ;
        or        di,di
        jg        short @1@422
    ;
    ;             {
    ;               y = 1;
    ;
        mov       di,1
    ;
    ;               dy = 1;
    ;
        mov       word ptr [bp-4],1
@1@422:
        call      near ptr _tstKbd
        or        ax,ax
        jne       @@0
        jmp       @1@58
@@0:
    ;
    ;             }
    ;
    ;
    ;       }
    ;
    ;       PutScr(SaveScr);
    ;
        push      ss
        lea       ax,word ptr [bp-4012]
        push      ax
        call      near ptr _PutScr
        pop       cx
        pop       cx
    ;
    ;       GotoXY(savey, savex);
    ;
        push      word ptr [bp-10]
        push      word ptr [bp-12]
        call      near ptr _GotoXY
        pop       cx
        pop       cx
    ;
    ;       return 0;
    ;
        xor       ax,ax
        jmp       short @1@478
@1@478:
    ;
    ;   }
    ;
```

```
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret
_main   endp

_TEXT   ends

_DATA   segment word public 'DATA'
s@      label   byte
        db      'Press any key to quit'
        db      0
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
        public  _main
        extrn   _PutStr:near
        extrn   _PutChar:near
        extrn   _PutScr:near
        extrn   _Capture:near
        extrn   _tstKbd:near
        extrn   _ClrScrn:near
        extrn   _GetXY:near
        extrn   _GotoXY:near
_s@     equ     s@
        end
```

11.11.3 EX11_1a.asm

```
; Assembly code to link with a C/C++ program.
; This code directly manipulates the screen giving C++
; direct access control of the screen.
;
; Note: Like PGM11_1.ASM, this code is relatively inefficient.
; It could be sped up quite a bit using the 80x86 string instructions.
; However, its inefficiency is actually a plus here since we don't
; want the C/C++ program (Ex11_1.cpp) running too fast anyway.
;
;
; This code assumes that Ex11_1.cpp is compiled using the LARGE
; memory model (far procs and far pointers).

                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

                .386                    ;Comment out these two statements
                option   segment:use16 ; if you are not using an 80386.

; ScrSeg- This is the video screen's segment address.  It should be
;               B000 for mono screens and B800 for color screens.

ScrSeg          =       0B800h

_TEXT           segment para public 'CODE'
                assume  cs:_TEXT

; _Capture-      Copies the data on the screen to the array passed
;                by reference as a parameter.
;
; procedure Capture(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y := 0 to 24 do
;           for x := 0 to 79 do
```

```
;                SCREEN[y,x] := ScrCopy[y,x];
; end;
;
;
; Activation record for Capture:
;
;       |                        |
;       | Previous stk contents  |
;       -------------------------
;       |  ScrCopy Seg Adrs      |
;       --                      --
;       | ScrCopy offset Adrs    |
;       -------------------------
;       | Return Adrs (offset)   |
;       -------------------------
;       |  X coordinate value    |
;       -------------------------
;       |  Y coordinate value    |
;       -------------------------
;       | Registers, etc.        |
;       ------------------------- <- SP


ScrCopy_cap     textequ  <dword ptr [bp+4]>
X_cap           textequ  <word ptr [bp-2]>
Y_cap           textequ  <word ptr [bp-4]>

                public   _Capture
_Capture        proc     near
                push     bp
                mov      bp, sp

                push     es
                push     ds
                push     si
                push     di
                pushf
                cld

                mov      si, ScrSeg             ;Set up pointer to SCREEN
                mov      ds, si                 ; memory (ScrSeg:0).
                sub      si, si

                les      di, ScrCopy_cap        ;Get ptr to capture array.

                mov      cx, 1000               ;4000 dwords on the screen
        rep     movsd

                popf
                pop      di
                pop      si
                pop      ds
                pop      es
                mov      sp, bp
                pop      bp
                ret
_Capture        endp


; _PutScr-      Copies array passed by reference onto the screen.
;
; procedure PutScr(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y := 0 to 24 do
;           for x := 0 to 79 do
;                 ScrCopy[y,x] := SCREEN[y,x];
; end;
;
```

```
;
; Activation record for PutScr:
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |  ScrCopy Seg Adrs     |
;       --                     --
;       | ScrCopy offset Adrs   |
;       -------------------------
;       | Return Adrs (offset)  |
;       -------------------------
;       |     BP Value          | <- BP
;       -------------------------
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       | Registers, etc.       |
;       ------------------------- <- SP


ScrCopy_fill    textequ  <dword ptr [bp+4]>
X_fill          textequ  <word ptr [bp-2]>
Y_fill          textequ  <word ptr [bp-4]>

                public  _PutScr
_PutScr         proc    near
                push     bp
                mov      bp, sp

                push     es
                push     ds
                push     si
                push     di
                pushf
                cld

                mov      di, ScrSeg             ;Set up pointer to SCREEN
                mov      es, di                 ; memory (ScrSeg:0).
                sub      di, di

                lds      si, ScrCopy_cap        ;Get ptr to capture array.

                mov      cx, 1000               ;1000 dwords on the screen
        rep     movsd

                popf
                pop      di
                pop      si
                pop      ds
                pop      es
                mov      sp, bp
                pop      bp
                ret
_PutScr         endp


; GotoXY-Positions the cursor at the specified X, Y coordinate.
;
; procedure gotoxy(y,x:integer);
; begin
;       BIOS(posnCursor,x,y);
; end;
;
; Activation record for GotoXY
;
```

```
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


X_gxy           textequ <byte ptr [bp+6]>
Y_gxy           textequ <byte ptr [bp+4]>

                public  _GotoXY
_GotoXY         proc    near
                push    bp
                mov     bp, sp

                mov     ah, 2       ;Magic BIOS value for gotoxy.
                mov     bh, 0       ;Display page zero.
                mov     dh, Y_gxy   ;Set up BIOS (X,Y) parameters.
                mov     dl, X_gxy
                int     10h         ;Make the BIOS call.

                mov     sp, bp
                pop     bp
                ret
_GotoXY         endp


; ClrScrn-    Clears the screen and positions the cursor at (0,0).
;
; procedure ClrScrn;
; begin
;       BIOS(Initialize)
; end;
;
; Activation record for ClrScrn
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |  Return Adrs (offset) |
;       ------------------------- <- SP


                public  _ClrScrn
_ClrScrn        proc    near

                mov     ah, 6       ;Magic BIOS number.
                mov     al, 0       ;Clear entire screen.
                mov     bh, 07      ;Clear with black spaces.
                mov     cx, 0000    ;Upper left corner is (0,0)
                mov     dl, 79      ;Lower X-coordinate
                mov     dh, 24      ;Lower Y-coordinate
                int     10h         ;Make the BIOS call.

                push    0           ;Position the cursor to (0,0)
                push    0           ; after the call.
                call    _GotoXY
                add     sp, 4       ;Remove parameters from stack.

                ret
_ClrScrn        endp
```

```
; tstKbd-       Checks to see if a key is available at the keyboard.
;
; function tstKbd:boolean;
; begin
;       if BIOSKeyAvail then eat key and return true
;       else return false;
; end;
;
; Activation record for tstKbd
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |  Return Adrs (offset) |
;       ------------------------- <- SP

                public  _tstKbd
_tstKbd         proc    near
                mov     ah, 1           ;Check to see if key avail.
                int     16h
                je      NoKey
                mov     ah, 0           ;Eat the key if there is one.
                int     16h
                mov     ax, 1           ;Return true.
                ret

NoKey:          mov     ax, 0           ;No key, so return false.
                ret
_tstKbd         endp


; GetXY- Returns the cursor's current X and Y coordinates.
;
; procedure GetXY(var x:integer; var y:integer);
;
; Activation record for GetXY
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |    Y Coordinate       |
;       ---    Address        ---
;       |                       |
;       -------------------------
;       |    X coordinate       |
;       ---    Address        ---
;       |                       |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


GXY_X           textequ  <[bp+4]>
GXY_Y           textequ  <[bp+8]>

                public  _GetXY
_GetXY          proc    near
                push    bp
                mov     bp, sp
                push    es

                mov     ah, 3           ;Read X, Y coordinates from
                mov     bh, 0           ; BIOS
                int     10h
```

```
                les     bx, GXY_X
                mov     es:[bx], dl
                mov     byte ptr es:[bx+1], 0

                les     bx, GXY_Y
                mov     es:[bx], dh
                mov     byte ptr es:[bx+1], 0

                pop     es
                pop     bp
                ret
_GetXY          endp


; PutChar- Outputs a single character to the screen at the current
;          cursor position.
;
; procedure PutChar(ch:char);
;
; Activation record for PutChar
;
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       | char (in L.O. byte    |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


ch_pc           textequ  <[bp+4]>

                public  _PutChar
_PutChar        proc     near
                push     bp
                mov      bp, sp

                mov      al, ch_pc
                mov      ah, 0eh
                int      10h

                pop      bp
                ret
_PutChar        endp



; PutStr-  Outputs a string to the display at the current cursor position.
;          Note that a string is a sequence of characters that ends with
;          a zero byte.
;
; procedure PutStr(var str:string);
;
; Activation record for PutStr
;
```

```
;       |                       |
;       | Previous stk contents |
;       -------------------------
;       |      String           |
;       ---    Address        ---
;       |                       |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


Str_ps          textequ  <[bp+4]>

                public   _PutStr
_PutStr         proc     near
                push     bp
                mov      bp, sp
                push     es

                les      bx, Str_ps
PS_Loop:        mov      al, es:[bx]
                cmp      al, 0
                je       PC_Done

                push     ax
                call     _PutChar
                pop      ax
                inc      bx
                jmp      PS_Loop

PC_Done:        pop      es
                pop      bp
                ret
_PutStr         endp



; StartPgm-     This is where DOS starts running the program.  This is
;               a substitute for the C0L.OBJ file normally linked in by
;               the Borland C++ compiler.  This code provides this
;               routine to avoid legal problems (i.e., distributing
;               unlinked Borland libraries).  You can safely ignore
;               this code.  Note that the C++ main program is a near
;               procedure, so this code needs to be in the _TEXT segment.

                extern   _main:near
StartPgm        proc     near

                mov      ax, _DATA
                mov      ds, ax
                mov      es, ax
                mov      ss, ax
                lea      sp, EndStk

                call     near ptr _main
                mov      ah, 4ch
                int      21h
StartPgm        endp

_TEXT           ends

_DATA           segment  word public "DATA"
stack           word     1000h dup (?)
EndStk          word     ?
_DATA           ends
```

```
sseg            segment  para stack 'STACK'
                word     1000h dup (?)
sseg            ends
                end      StartPgm
```

11.12 Programming Projects

1) Write a version of the matrix multiply program inputs two 4x4 integer matrices from the user and compute their matrix product (see Chapter Eight question set). The matrix multiply algorithm (computing C := A * B) is

```
for i := 0 to 3 do
        for j := 0 to 3 do begin

                c[i,j] := 0;
                for k := 0 to 3 do
                    c[i,j] := c[i,j] + a[i,k] * b[k,j];
        end;
```

The program should have three procedures: InputMatrix, PrintMatrix, and MatrixMul. They have the following prototypes:

```
Procedure InputMatrix(var m:matrix);
procedure PrintMatrix(var m:matrix);
procedure MatrixMul(var result, A, B:matrix);
```

In particular note that these routines all pass their parameters by reference. Pass these parameters by reference on the stack.

Maintain all variables (e.g., i, j, k, etc.) on the stack using the techniques outlined in “Local Variable Storage” on page 604. In particular, do not keep the loop control variables in register.

Write a main program that makes appropriate calls to these routines to test them.

2) A pass by lazy evaluation parameter is generally a structure with three fields: a pointer to the thunk to call to the function that computes the value, a field to hold the value of the parameter, and a boolean field that contains false if the value field is uninitialized (the value field becomes initialized if the procedure writes to the value field or calls the thunk to obtain the value). Whenever the procedure writes a value to a pass by lazy evaluation parameter, it stores the value in the value field and sets the boolean field to true. Whenever a procedure wants to read the value, it first checks this boolean field. If it contains a true value, it simply reads the value from the value field; if the boolean field contains false, the procedure calls the thunk to compute the initial value. On return, the procedure stores the thunk result in the value field and sets the boolean field to true. Note that during any single activation of a procedure, the thunk for a parameter will be called, at most, one time. Consider the following Panacea procedure:

```
SampleEval: procedure(select:boolean; eval a:integer; eval b:integer);
var
        result:integer;
endvar;
begin SimpleEval;

        if (select) then

                result := a;

        else
                result := b;

        endif;
        writeln(result+2);

end SampleEval;
```

Write an assembly language program that implements SampleEval. From your main pro-

gram call SampleEval a couple of times passing it different thunks for the a and b parameters. Your thunks can simply return a single value when called.

3) Write a shuffle routine to which you pass an array of 52 integers by reference. The routine should fill the array with the values 1..52 and then randomly shuffle the items in the array. Use the Standard Library random and randomize routines to select an index in the array to swap. See Chapter Seven, "Random Number Generation: Random, Randomize" on page 343 for more details about the random function. Write a main program that passes an array to this procedure and prints out the result.

11.13 Summary

In an assembly language program, all you need is a call and ret instruction to implement procedures and functions. Chapter Seven covers the basic use of procedures in an 80x86 assembly language program; this chapter describes how to organize program units like procedures and functions, how to pass parameters, allocate and access local variables, and related topics.

This chapter begins with a review of what a procedure is, how to implement procedures with MASM, and the difference between near and far procedures on the 80x86. For details, see the following sections:

- "Procedures" on page 566
- "Near and Far Procedures" on page 568
- "Forcing NEAR or FAR CALLs and Returns" on page 568
- "Nested Procedures" on page 569

Functions are a very important construct in high level languages like Pascal. However, there really isn't a difference between a function and a procedure in an assembly language program. Logically, a function returns a result and a procedure does not; but you declare and call procedures and functions identically in an assembly language program. See

- "Functions" on page 572

Procedures and functions often produce *side effects*. That is, they modify the values of registers and non-local variables. Often, these side effects are undesirable. For example, a procedure may modify a register that the caller needs preserved. There are two basic mechanisms for preserving such values: callee preservation and caller preservation. For details on these preservation schemes and other important issues see

- "Saving the State of the Machine" on page 572
- "Side Effects" on page 602

One of the major benefits to using a procedural language like Pascal or C++ is that you can easily pass parameters to and from procedures and functions. Although it is a little more work, you can pass parameters to your assembly language functions and procedures as well. This chapter discusses how and where to pass parameters. It also discusses how to access the parameters inside a procedure or function. To read about this, see sections

- "Parameters" on page 574
- "Pass by Value" on page 574
- "Pass by Reference" on page 575
- "Pass by Value-Returned" on page 575
- "Pass by Name" on page 576
- "Pass by Lazy-Evaluation" on page 577
- "Passing Parameters in Registers" on page 578
- "Passing Parameters in Global Variables" on page 580
- "Passing Parameters on the Stack" on page 581
- "Passing Parameters in the Code Stream" on page 590
- "Passing Parameters via a Parameter Block" on page 598

Since assembly language doesn't really support the notion of a function, per se, implementing a function consists of writing a procedure with a return parameter. As such, function results are quite similar to parameters in many respects. To see the similarities, check out the following sections:

- "Function Results" on page 600
- "Returning Function Results in a Register" on page 601
- "Returning Function Results on the Stack" on page 601
- "Returning Function Results in Memory Locations" on page 602

Most high level languages provide *local variable storage* associated with the activation and deactivation of a procedure or function. Although few assembly language programs use local variables in an identical fashion, it's very easy to implement dynamic allocation of local variables on the stack. For details, see section

- "Local Variable Storage" on page 604

Recursion is another HLL facility that is very easy to implement in an assembly language program. This chapter discusses the technique of recursion and then presents a simple example using the Quicksort algorithm. See

- "Recursion" on page 606

11.14 Questions

1) Explain how the CALL and RET instructions operate.

2) What are the operands for the PROC assembler directive? What is their function?

3) Rewrite the following code using PROC and ENDP:

```
       FillMem:          moval, 0FFh
FillLoop:                mov[bx], al
                         incbx
                         loop FillLoop
                         ret
```

4) Modify your answer to problem (3) so that all affected registers are preserved by the FillMem procedure.

5) What happens if you fail to put a transfer of control instruction (such as a JMP or RET) immediately before the ENDP directive in a procedure?

6) How does the assembler determine if a CALL is near or far? How does it determine if a RET instruction is near or far?

7) How can you override the assembler's default decision whether to use a near or far CALL or RET?

8) Is there ever a need for nested procedures in an assembly language program? If so, give an example.

9) Give an example of why you might want to nest a segment inside a procedure.

10) What is the difference between a function, and a procedure?

11) Why should subroutines preserve the registers that they modify?

12) What are the advantages and disadvantages of caller-preserved values and callee-preserved values?

13) What are parameters?

14) How do the following parameter passing mechanisms work?

    a) Pass by value

    b) Pass by reference

    c) Pass by value-returned

    d) Pass by name

15) Where is the best place to pass parameters to a procedure?

16) List five different locations / methods for passing parameters to or from a procedure.

17) How are parameters that are passed on the stack accessed within a procedure?

18) What's the best way to deallocate parameters passed on the stack when the procedure terminates execution?

19) Given the following Pascal procedure definition:

```
procedure PascalProc(i,j,k:integer);
```

    Explain how you would access the parameters of an equivalent assembly language program, assuming that the procedure is a near procedure.

20) Repeat problem (19) assuming that the procedure is a far procedure.

21) What does the stack look like during the execution of the procedure in problem (19)? Problem (20)?

22) How does an assembly language procedure gain access to parameters passed in the code stream?

23) How does the 80x86 skip over parameters passed in the code stream and continue program execution beyond them when the procedure returns to the caller?

24) What is the advantage to passing parameters via a parameter block?

25) Where are function results typically returned?

26) What is a side effect?

27) Where are local (temporary) variables typically allocated?

28) How do you allocate local (temporary) variables within a procedure?

29) Assuming you have three parameters passed by value on the stack and 4 different local variables, what does the activation record look like after the local variables have been allocated (assume a near procedure and no registers other than BP have been pushed onto the stack).

30) What is recursion?

Procedures: Advanced Topics Chapter 12

The last chapter described how to create procedures, pass parameters, and allocate and access local variables. This chapter picks up where that one left off and describes how to access non-local variables in other procedures, pass procedures as parameters, and implement some user-defined control structures.

12.0 Chapter Overview

This chapter completes the discussion of procedures, parameters, and local variables begun in the previous chapter. This chapter describes how block structured languages like Pascal, Modula-2, Algol, and Ada access local and non-local variables. This chapter also describes how to implement a user-defined control structure, the *iterator*. Most of the material in this chapter is of interest to compiler writers and those who want to learn how compilers generate code for certain types of program constructs. Few pure assembly language programs will use the techniques this chapter describes. Therefore, none of the material in this chapter is particularly important to those who are just learning assembly language. However, if you are going to write a compiler, or you want to learn how compilers generate code so you can write efficient HLL programs, you will want to learn the material in this chapter sooner or later.

This chapter begins by discussing the notion of *scope* and how HLLs like Pascal access variables in nested procedures. The first section discusses the concept of lexical nesting and the use of static links and displays to access non-local variables. Next, this chapter discusses how to pass variables at different lex levels as parameters. The third section discusses how to pass parameters of one procedure as parameters to another procedure. The fourth major topic this chapter covers is passing procedures as parameters. This chapter concludes with a discussion of *iterators*, a user-defined control structure.

This chapter assumes a familiarity with a block structured language like Pascal or Ada. If your only HLL experience is with a non-block structured language like C, C++, BASIC, or FORTRAN, some of the concepts in this chapter may be completely new and you will have trouble understanding them. Any introductory text on Pascal or Ada will help explain any concept you don't understand that this chapter assumes is a prerequisite.

12.1 Lexical Nesting, Static Links, and Displays

In block structured languages like Pascal[1] it is possible to *nest* procedures and functions. Nesting one procedure within another limits the access to the nested procedure; you cannot access the nested procedure from outside the enclosing procedure. Likewise, variables you declare within a procedure are visible inside that procedure and to all procedures nested within that procedure[2]. This is the standard block structured language notion of *scope* that should be quite familiar to anyone who has written Pascal or Ada programs.

There is a good deal of complexity hidden behind the concept of scope, or lexical nesting, in a block structured language. While accessing a local variable in the current activation record is efficient, accessing global variables in a block structured language can be very inefficient. This section will describe how a HLL like Pascal deals with non-local identifiers and how to access global variables and call non-local procedures and functions.

1. Note that C and C++ are not block structured languages. Other block structured languages include Algol, Ada, and Modula-2.
2. Subject, of course, to the limitation that you not reuse the identifier within the nested procedure.

12.1.1 Scope

Scope in most high level languages is a static, or compile-time concept[3]. Scope is the notion of when a name is visible, or accessible, within a program. This ability to hide names is useful in a program because it is often convenient to reuse certain (non-descriptive) names. The i variable used to control most for loops in high level languages is a perfect example. Throughout this chapter you've seen equates like xyz_i, xyz_j, etc. The reason for choosing such names is that MASM doesn't support the same notion of scoped names as high level languages. Fortunately, MASM 6.x and later *does* support scoped names.

By default, MASM 6.x treats statement labels (those with a colon after them) as local to a procedure. That is, you may only reference such labels within the procedure in which they are declared. *This is true even if you nest one procedure inside another.* Fortunately, there is no good reason why anyone would want to nest procedures in a MASM program.

Having local labels within a procedure is nice. It allows you to reuse statement labels (e.g., loop labels and such) without worrying about name conflicts with other procedures. Sometimes, however, you may want to turn off the scoping of names in a procedure; a good example is when you have a case statement whose jump table appears outside the procedure. If the case statement labels are local to the procedure, they will not be visible outside the procedure and you cannot use them in the case statement jump table (see "CASE Statements" on page 525). There are two ways you can turn off the scoping of labels in MASM 6.x. The first way is to include the statement in your program:

```
option      noscoped
```

This will turn off variable scoping from that point forward in your program's source file. You can turn scoping back on with a statement of the form

```
option      scoped
```

By placing these statements around your procedure you can selectively control scoping.

Another way to control the scoping of individual names is to place a double colon ("::") after a label. This informs the assembler that this particular name should be global to the enclosing procedure.

MASM, like the C programming language, supports three levels of scope: public, global (or static), and local. Local symbols are visible only within the procedure they are defined. Global symbols are accessible throughout a source file, but are not visible in other program modules. Public symbols are visible throughout a program, across modules. MASM uses the following default scoping rules:

- By default, statement labels appearing in a procedure are local to that procedure.
- By default, all procedure names are public.
- By default, most other symbols are global.

Note that these rules apply to MASM 6.x only. Other assemblers and earlier versions of MASM follow different rules.

Overriding the default on the first rule above is easy – either use the option noscoped statement or use a double colon to make a label global. You should be aware, though, that you cannot make a local label public using the public or externdef directives. You must make the symbol global (using either technique) before you make it public.

Having all procedure names public by default usually isn't much of a problem. However, it might turn out that you want to use the same (local) procedure name in several different modules. If MASM automatically makes such names public, the linker will give you an error because there are multiple public procedures with the same name. You can turn on and off this default action using the following statements:

```
option      proc:private          ;procedures are global
```

3. There are languages that support dynamic, or run-time, scope; this text will not consider such languages.

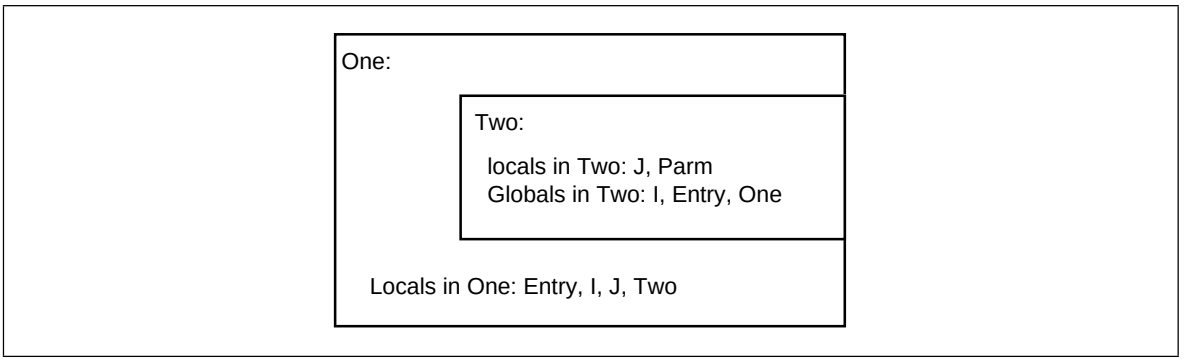


Figure 12.1 Identifier Scope

```
option      proc:export          ;procedures are public
```

Note that some debuggers only provide symbolic information if a procedure's name is public. This is why MASM 6.x defaults to public names. This problem does not exist with CodeView; so you can use whichever default is most convenient. Of course, if you elect to keep procedure names private (global only), then you will need to use the public or externdef directive to make desired procedure names public.

This discussion of local, global, and public symbols applies *mainly* to statement and procedure labels. It does *not* apply to variables you've declared in your data segment, equates, macros, typedefs, or most other symbols. Such symbols are always global regardless of where you define them. The only way to make them public is to specify their names in a public or externdef directive.

There is a way to declare parameter names and local variables, allocated on the stack, such that their names are local to a given procedure. See the proc directive in the MASM reference manual for details.

The scope of a name limits its visibility within a program. That is, a program has access to a variable name only within that name's scope. Outside the scope, the program cannot access that name. Many programming languages, like Pascal and C++, allow you to reuse identifiers if the scopes of those multiple uses do not overlap. As you've seen, MASM provides some minimal scoping features for statement labels. There is, however, another issue related to scope: *address binding* and *variable lifetime*. Address binding is the process of associating a memory address with a variable name. Variable lifetime is that portion of a program's execution during which a memory location is bound to a variable. Consider the following Pascal procedures:

```
procedure One(Entry:integer);
var
       i,j:integer;

       procedure Two(Parm:integer);
       var j:integer;
       begin

                 for j:= 0 to 5 do writeln(i+j);
                 if Parm < 10 then One(Parm+1);

       end;

begin {One}
       for i := 1 to 5 do Two(Entry);
end;
```

Figure 12.1 shows the scope of identifiers One, Two, Entry, i, j, and Parm.

The local variable j in Two masks the identifier j in procedure One while inside Two.

12.1.2 Unit Activation, Address Binding, and Variable Lifetime

Unit activation is the process of calling a procedure or function. The combination of an activation record and some executing code is considered an *instance* of a routine. When unit activation occurs a routine binds machine addresses to its local variables. Address binding (for local variables) occurs when the routine adjusts the stack pointer to make room for the local variables. The lifetime of those variables is from that point until the routine destroys the activation record eliminating the local variable storage.

Although scope limits the visibility of a name to a certain section of code and does not allow duplicate names within the same scope, this does not mean that there is only one address bound to a name. It is quite possible to have several addresses bound to the same name at the same time. Consider a recursive procedure call. On each activation the procedure builds a new activation record. Since the previous instance still exists, there are now two activation records on the stack containing local variables for that procedure. As additional recursive activations occur, the system builds more activation records each with an address bound to the same name. To resolve the possible ambiguity (which address do you access when operating on the variable?), the system always manipulates the variable in the most recent activation record.

Note that procedures One and Two in the previous section are *indirectly recursive*. That is, they both call routines which, in turn, call themselves. Assuming the parameter to One is less than 10 on the initial call, this code will generate multiple activation records (and, therefore, multiple copies of the local variables) on the stack. For example, were you to issue the call One(9), the stack would look like Figure 12.2 upon first encountering the end associated with the procedure Two.

As you can see, there are several copies of I and J on the stack at this point. Procedure Two (the currently executing routine) would access J in the most recent activation record that is at the bottom of Figure 12.2. The previous instance of Two will only access the variable J in its activation record when the current instance returns to One and then back to Two.

The lifetime of a variable's instance is from the point of activation record creation to the point of activation record destruction. Note that the first instance of J above (the one at the top of the diagram above) has the longest lifetime and that the lifetimes of all instances of J overlap.

12.1.3 Static Links

Pascal will allow procedure Two access to I in procedure One. However, when there is the possibility of recursion there may be several instances of i on the stack. Pascal, of course, will only let procedure Two access the most recent instance of i. In the stack diagram in Figure 12.2, this corresponds to the value of i in the activation record that begins with "One(9+1) parameter." The only problem is *how do you know where to find the activation record containing i?*

A quick, but poorly thought out answer, is to simply index backwards into the stack. After all, you can easily see in the diagram above that i is at offset eight from Two's activation record. Unfortunately, this is not always the case. Assume that procedure Three also calls procedure Two and the following statement appears within procedure One:

```
If (Entry <5) then Three(Entry*2) else Two(Entry);
```

With this statement in place, it's quite possible to have two different stack frames upon entry into procedure Two: one with the activation record for procedure Three sandwiched between One and Two's activation records and one with the activation records for procedures One and Two adjacent to one another. Clearly a fixed offset from Two's activation record will not always point at the i variable on One's most recent activation record.

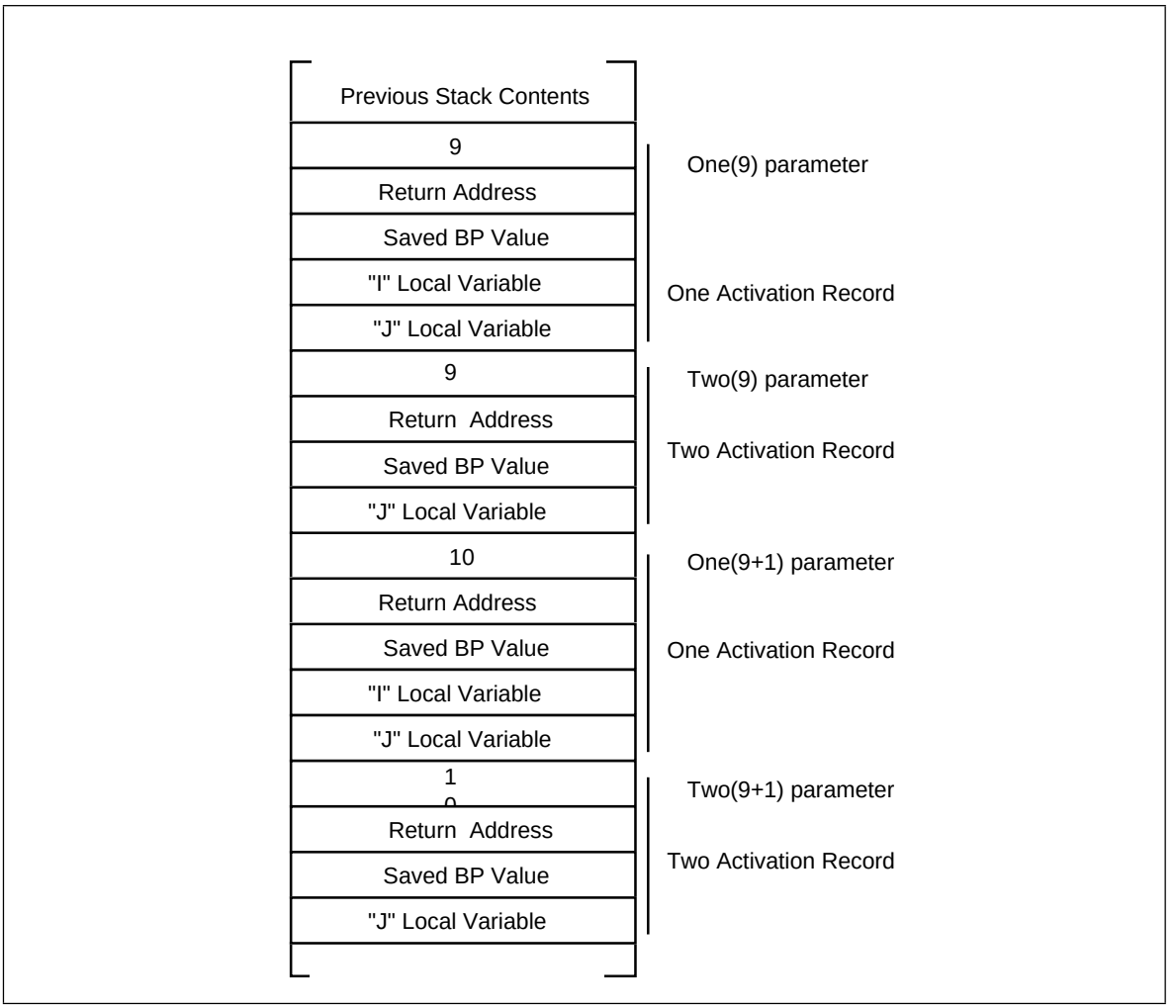


Figure 12.2 Indirect Recursion

The astute reader might notice that the saved bp value in Two's activation record points at the caller's activation record. You might think you could use this as a pointer to One's activation record. But this scheme fails for the same reason the fixed offset technique fails. Bp's old value, the *dynamic link*, points at the caller's activation record. Since the caller isn't necessarily the enclosing procedure the dynamic link might not point at the enclosing procedure's activation record.

What is really needed is a pointer to the enclosing procedure's activation record. Many compilers for block structured languages create such a pointer, the *static link*. Consider the following Pascal code:

```
procedure Parent;
var i,j:integer;

      procedure Child1;
      var j:integer;
      begin

                for j := 0 to 2 do writeln(i);

      end {Child1};

      procedure Child2;
      var i:integer;
      begin

                for i := 0 to 1 do Child1;

      end {Child2};
```

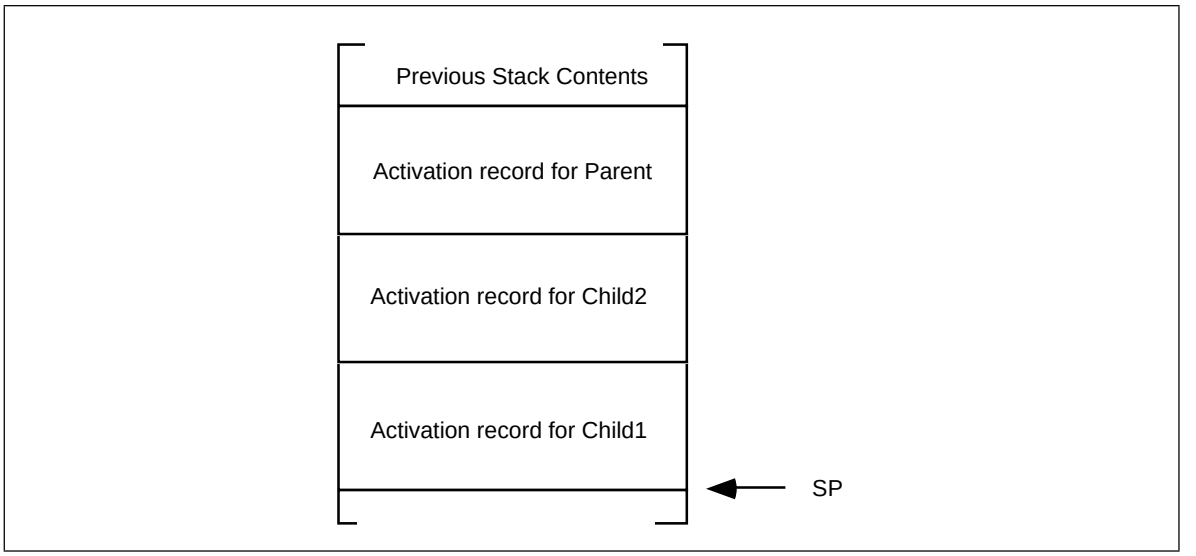


Figure 12.3 Activation Records after Several Nested Calls

```
begin {Parent}
       Child2;
       Child1;
end;
```

Just after entering Child1 for the first time, the stack would look like Figure 12.3. When Child1 attempts to access the variable i from Parent, it will need a pointer, the static link, to Parent's activation record. Unfortunately, there is no way for Child1, upon entry, to figure out on it's own where Parent's activation record lies in memory. It will be necessary for the caller (Child2 in this example) to pass the static link to Child1. In general, the callee can treat the static link as just another parameter; usually pushed on the stack immediately before executing the call instruction.

To fully understand how to pass static links from call to call, you must first understand the concept of a lexical level. Lexical levels in Pascal correspond to the static nesting levels of procedures and functions. Most compiler writers specify lex level zero as the main program. That is, all symbols you declare in your main program exist at lex level zero. Procedure and function names appearing in your main program define lex level one, *no matter how many procedures or functions appear in the main program.* They all begin a new copy of lex level one. For each level of nesting, Pascal introduces a new lex level. Figure 12.4 shows this.

During execution, a program may only access variables at a lex level less than or equal to the level of the current routine. Furthermore, only one set of values at any given lex level are accessible at any one time[4] and those values are always in the most recent activation record at that lex level.

Before worrying about how to access non-local variables using a static link, you need to figure out how to pass the static link as a parameter. When passing the static link as a parameter to a program unit (procedure or function), there are three types of calling sequences to worry about:

- A program unit calls a *child* procedure or function. If the current lex level is *n*, then a child procedure or function is at lex level *n+1* and is local to

4. There is one exception. If you have a *pointer* to a variable and the pointer remains accessible, you can access the data it points at even if the variable actually holding that data is inaccessible. Of course, in (standard) Pascal you cannot take the address of a local variable and put it into a pointer. However, certain dialects of Pascal (e.g., Turbo) and other block structured languages will allow this operation.

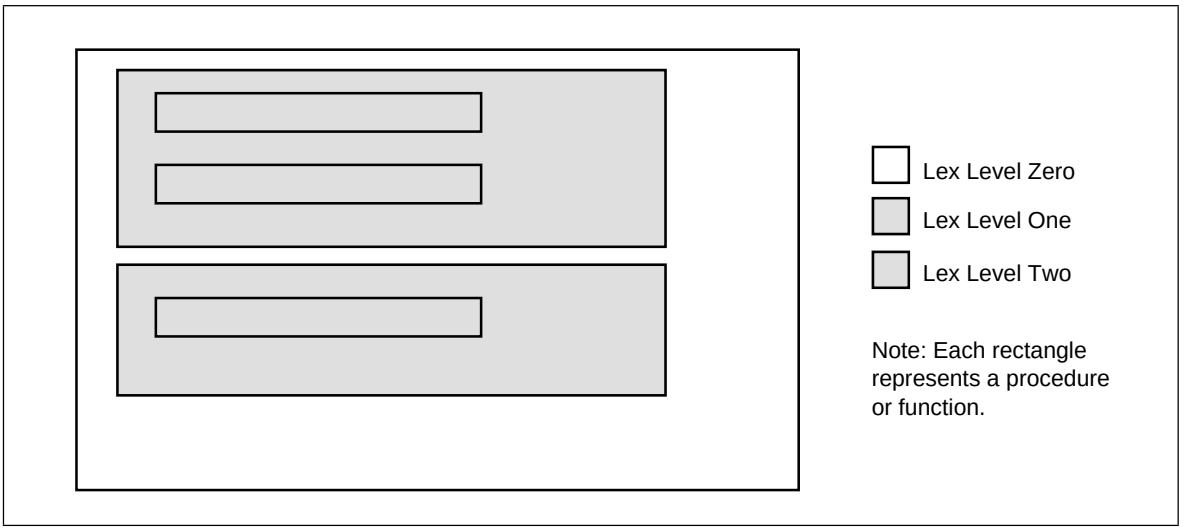


Figure 12.4 Procedure Schematic Showing Lexical Levels

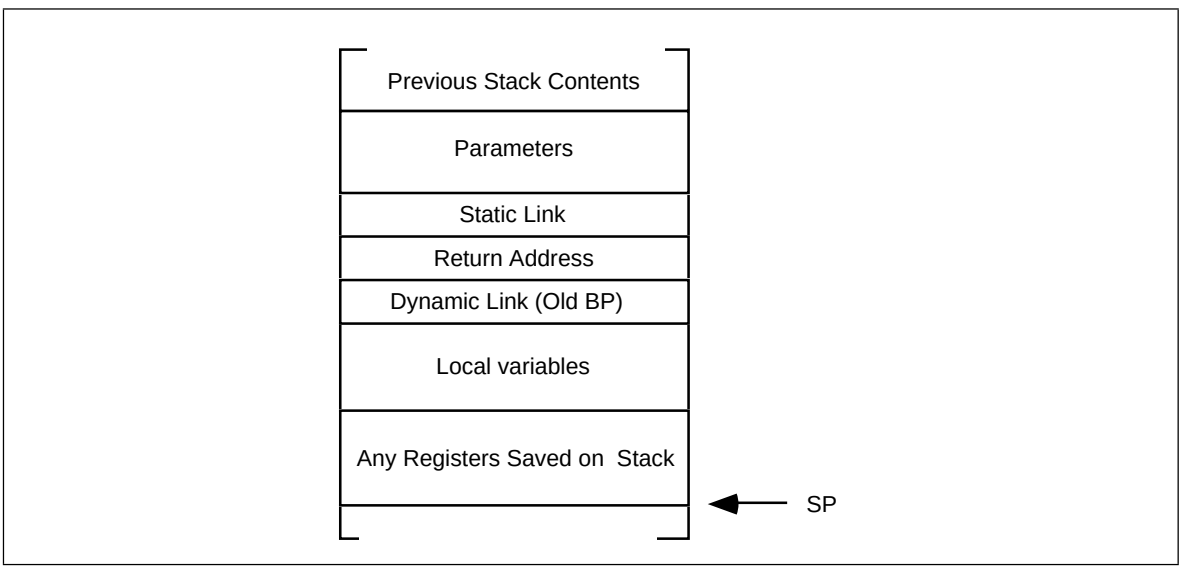


Figure 12.5 Generic Activation Record

the current program unit. Note that most block structured languages do not allow calling procedures or functions at lex levels greater than *n+1*.

- A program unit calls a *peer* procedure or function. A peer procedure or function is one at the same lexical level as the current caller and a single program unit encloses both program units.
- A program unit calls an *ancestor* procedure or function. An ancestor unit is either the parent unit, a parent of an ancestor unit, or a peer of an ancestor unit.

Calling sequences for the first two types of calls above are very simple. For the sake of this example, assume the activation record for these procedures takes the generic form in Figure 12.5.

When a parent procedure or function calls a child program unit, the static link is nothing more than the value in the bp register immediately prior to the call. Therefore, to pass the static link to the child unit, just push bp before executing the call instruction:

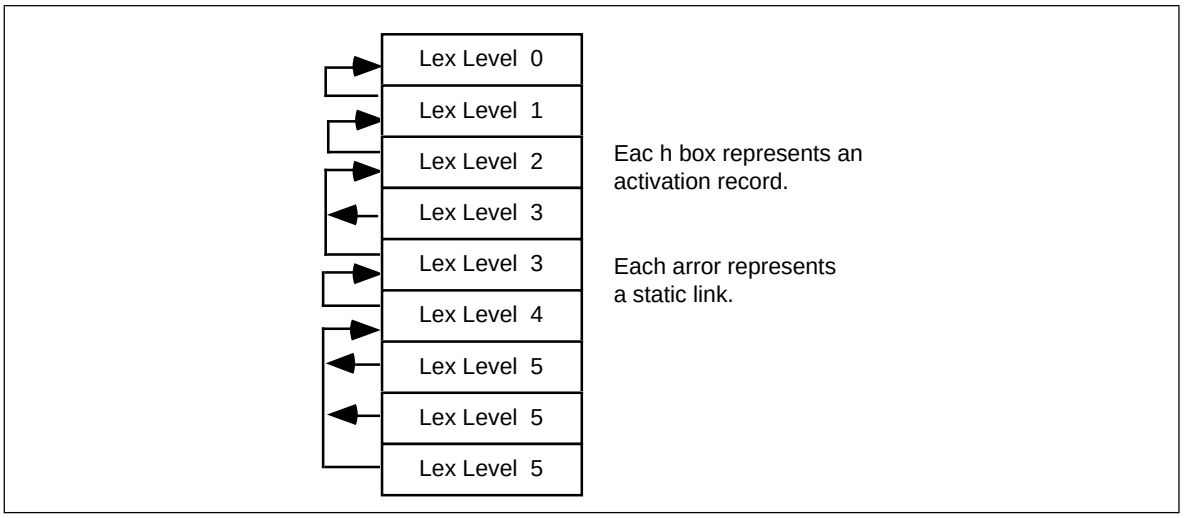


Figure 12.6 Static Links

```
<Push Other Parameters onto the stack>
          push      bp
          call      ChildUnit
```

Of course the child unit can process the static link off the stack just like any other parameter. In this case, that the static and dynamic links are exactly the same. In general, however, this is not true.

If a program unit calls a peer procedure or function, the current value in bp is not the static link. It is a pointer to the caller's local variables and the peer procedure cannot access those variables. However, as peers, the caller and callee share the same parent program unit, so the caller can simply push a copy of its static link onto the stack before calling the peer procedure or function. The following code will do this, assuming all procedures and functions are near:

```
<Push Other Parameters onto the Stack>
          push      [bp+4]                  ;Push static link onto stk.
          call      PeerUnit
```

If the procedure or function is far, the static link would be two bytes farther up the stack, so you would need to use the following code:

```
<Push Other Parameters onto the Stack>
          push      [bp+6]                  ;Push static link onto stk.
          call      PeerUnit
```

Calling an ancestor is a little more complex. If you are currently at lex level *n* and you wish to call an ancestor at lex level *m* (*m* < *n*), you will need to *traverse* the list of static links to find the desired activation record. The static links form a *list* of activation records. By following this chain of activation records until it ends, you can step through the most recent activation records of all the enclosing procedures and functions of a particular program unit. The stack diagram in Figure 12.6 shows the static links for a sequence of procedure calls statically nested five lex levels deep.

If the program unit currently executing at lex level five wishes to call a procedure at lex level three, it must push a static link to the most recently activated program unit at lex level two. In order to find this static link you will have to *traverse* the chain of static links. If you are at lex level *n* and you want to call a procedure at lex level *m* you will have to traverse (*n*-*m*)+1 static links. The code to accomplish this is

```
; Current lex level is 5. This code locates the static link for,
; and then calls a procedure at lex level 2. Assume all calls are
; near:

        <Push necessary parameters>

                mov     bx, [bp+4]    ;Traverse static link to LL 4.
                mov     bx, ss:[bx+4] ;To Lex Level 3.
                mov     bx, ss:[bx+4] ;To Lex Level 2.
                push    ss:[bx+4]     ;Ptr to most recent LL1 A.R.
                call    ProcAtLL2
```

Note the ss: prefix in the instructions above. Remember, the activation records are all in the stack segment and bx indexes the data segment by default.

12.1.4 Accessing Non-Local Variables Using Static Links

In order to access a non-local variable, you must traverse the chain of static links until you get a pointer to the desired activation record. This operation is similar to locating the static link for a procedure call outlined in the previous section, except you traverse only *n-m* static links rather than *(n-m)+1* links to obtain a pointer to the appropriate activation record. Consider the following Pascal code:

```
procedure Outer;
var i:integer;

        procedure Middle;
        var j:integer;

                procedure Inner;
                var k:integer;
                begin

                        k := 3;
                        writeln(i+j+k);

                end;

        begin {middle}

                j := 2;
                writeln(i+j);
                Inner;

        end; {middle}

begin {Outer}

        i := 1;
        Middle;

end; {Outer}
```

The Inner procedure accesses global variables at lex level *n-1* and *n-2* (where *n* is the lex level of the Inner procedure). The Middle procedure accesses a single global variable at lex level *m-1* (where *m* is the lex level of procedure Middle). The following assembly language code could implement these three procedures:

```
Outer           proc    near
                push    bp
                mov     bp, sp
                sub     sp, 2                   ;Make room for I.

                mov     word ptr [bp-2],1       ;Set I to one.
                push    bp                      ;Static link for Middle.
                call    Middle

                mov     sp, bp                  ;Remove local variables.
                pop     bp
                ret     2                       ;Remove static link on ret.
Outer           endp

Middle          proc    near
```

```
                        push    bp                      ;Save dynamic link
                        mov     bp, sp                  ;Set up activation record.
                        sub     sp, 2                   ;Make room for J.

                        mov     word ptr [bp-2],2       ;J := 2;
                        mov     bx, [bp+4]              ;Get static link to prev LL.
                        mov     ax, ss:[bx-2]           ;Get I's value.
                        add     ax, [bp-2]              ;Add to J and then
                        puti                            ; print the sum.
                        putcr
                        push    bp                      ;Static link for Inner.
                        call    Inner

                        mov     sp, bp
                        pop     bp
                        ret     2                       ;Remove static link on RET.
Middle                  endp

Inner                   proc    near
                        push    bp                      ;Save dynamic link
                        mov     bp, sp                  ;Set up activation record.
                        sub     sp, 2                   ;Make room for K.

                        mov     word ptr [bp-2],2       ;K := 3;
                        mov     bx, [bp+4]              ;Get static link to prev LL.
                        mov     ax, ss:[bx-2]           ;Get J's value.
                        add     ax, [bp-2]              ;Add to K

                        mov     bx, ss:[bx+4]           ;Get ptr to Outer's Act Rec.
                        add     ax, ss:[bx-2]           ;Add in I's value and then
                        puti                            ; print the sum.
                        putcr

                        mov     sp, bp
                        pop     bp
                        ret     2                       ;Remove static link on RET.
Inner                   endp
```

As you can see, accessing global variables can be very inefficient[5].

Note that as the difference between the activation records increases, it becomes less and less efficient to access global variables. Accessing global variables in the previous activation record requires only one additional instruction per access, at two lex levels you need two additional instructions, etc. If you analyze a large number of Pascal programs, you will find that most of them do not nest procedures and functions and in the ones where there are nested program units, they rarely access global variables. There is one major exception, however. Although Pascal procedures and functions rarely access local variables inside other procedures and functions, they frequently access global variables declared in the main program. Since such variables appear at lex level zero, access to such variables would be as inefficient as possible when using the static links. To solve this minor problem, most 80x86 based block structured languages allocate variables at lex level zero directly in the data segment and access them directly.

12.1.5 The Display

After reading the previous section you might get the idea that one should never use non-local variables, or limit non-local accesses to those variables declared at lex level zero. After all, it's often easy enough to put all shared variables at lex level zero. If you are designing a programming language, you can adopt the C language designer's philosophy and simply not provide block structure. Such compromises turn out to be unnecessary. There is a data structure, the *display*, that provides efficient access to *any* set of non-local variables.

5. Indeed, perhaps one of the main reasons the C programming language is not block structured is because the language designers wanted to avoid inefficient access to non-local variables.

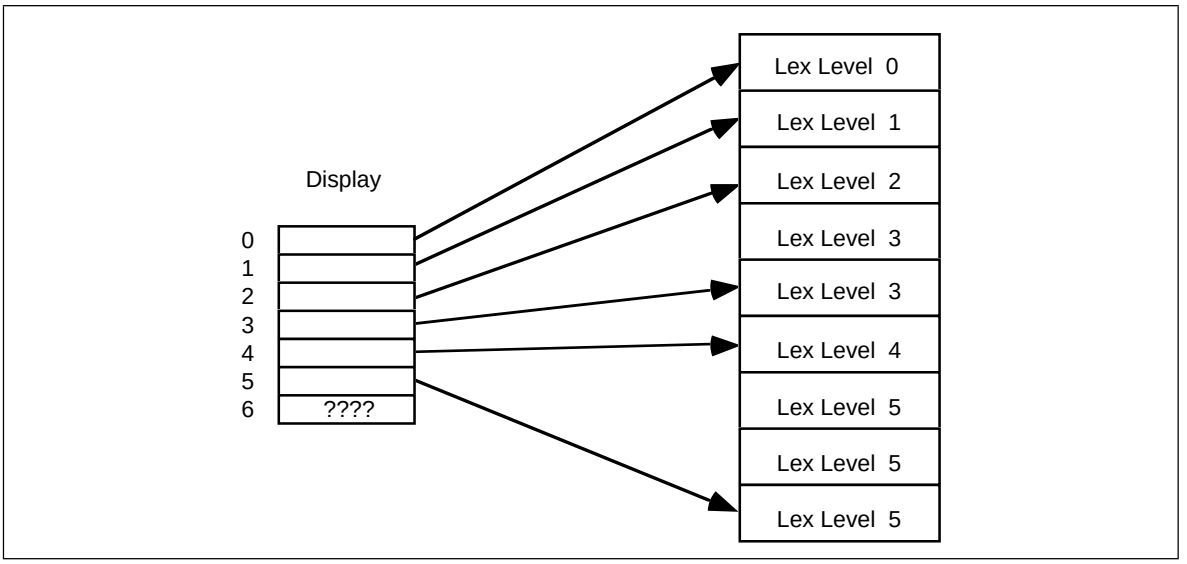


Figure 12.7 The Display

A display is simply an array of pointers to activation records. Display[0] contains a pointer to the most recent activation record for lex level zero, Display[1] contains a pointer to the most recent activation record for lex level one, and so on. Assuming you've maintained the Display array in the current data segment (always a good place to keep it) it only takes two instructions to access any non-local variable. Pictorially, the display works as shown in Figure 12.7.

Note that the entries in the display always point at the most recent activation record for a procedure at the given lex level. If there is no active activation record for a particular lex level (e.g., lex level six above), then the entry in the display contains garbage.

The maximum lexical nesting level in your program determines how many elements there must be in the display. Most programs have only three or four nested procedures (if that many) so the display is usually quite small. Generally, you will rarely require more than 10 or so elements in the display.

Another advantage to using a display is that each individual procedure can maintain the display information itself, the caller need not get involved. When using static links the calling code has to compute and pass the appropriate static link to a procedure. Not only is this slow, but the code to do this must appear before every call. If your program uses a display, the callee, rather than the caller, maintains the display so you only need one copy of the code per procedure. Furthermore, as the next example shows, the code to handle the display is short and fast.

Maintaining the display is very easy. Upon initial entry into a procedure you must first save the contents of the display array at the current lex level and then store the pointer to the current activation record into that same spot. Accessing a non-local variable requires only two instructions, one to load an element of the display into a register and a second to access the variable. The following code implements the Outer, Middle, and Inner procedures from the static link examples.

```
; Assume Outer is at lex level 1, Middle is at lex level 2, and
; Inner is at lex level 3. Keep in mind that each entry in the
; display is two bytes. Presumably, the variable Display is defined
; in the data segment.

Outer           proc    near
                push    bp
                mov     bp, sp
                push    Display[2]          ;Save current Display Entry
                sub     sp, 2               ;Make room for I.
```

```
                mov     word ptr [bp-4],1       ;Set I to one.
                call    Middle

                add     sp, 2                   ;Remove local variables
                pop     Display[2]              ;Restore previous value.
                pop     bp
                ret
Outer           endp

Middle          proc    near
                push    bp                      ;Save dynamic link.
                mov     bp, sp                  ;Set up our activation
record.
                push    Display[4]              ;Save old Display value.
                sub     sp, 2                   ;Make room for J.

                mov     word ptr [bp-2],2       ;J := 2;
                mov     bx, Display[2]          ;Get static link to prev LL.
                mov     ax, ss:[bx-4]           ;Get I's value.
                add     ax, [bp-2]              ;Add to J and then
                puti                            ; print the sum.
                putcr
                call    Inner

                add     sp, 2                   ;Remnove local variable.
                pop     Display[4]              ;Restore old Display value.
                pop     bp
                ret
Middle          endp

Inner           proc    near
                push    bp                      ;Save dynamic link
                mov     bp, sp                  ;Set up activation record.
                push    Display[6]              ;Save old display value
                sub     sp, 2                   ;Make room for K.

                mov     word ptr [bp-2],2       ;K := 3;
                mov     bx, Display[4]          ;Get static link to prev LL.
                mov     ax, ss:[bx-4]           ;Get J's value.
                add     ax, [bp-2]              ;Add to K

                mov     bx, Display[2]          ;Get ptr to Outer's Act Rec.
                add     ax, ss:[bx-4]           ;Add in I's value and then
                puti                            ; print the sum.
                putcr

                add     sp, 2
                pop     Display [6]
                pop     bp
                ret
Inner           endp
```

Although this code doesn't look particularly better than the former code, using a display is often much more efficient than using static links.

12.1.6 The 80286 ENTER and LEAVE Instructions

When designing the 80286, Intel's CPU designers decided to add two instructions to help maintain displays. Unfortunately, although their design works, is very general, and only requires data in the stack segment, it is very slow; much slower than using the techniques in the previous section. Although many non-optimizing compilers use these instructions, the best compilers avoid using them, if possible.

The leave instruction is very simple to understand. It performs the same operation as the two instructions:

```
                mov     sp, bp
                pop     bp
```

Therefore, you may use the instruction for the standard procedure exit code if you have an 80286 or later microprocessor. On an 80386 or earlier processor, the leave instruction is

shorter and faster than the equivalent move and pop sequence. However, the leave instruction is slower on 80486 and later processors.

The enter instruction takes two operands. The first is the number of bytes of local storage the current procedure requires, the second is the lex level of the current procedure. The enter instruction does the following:

```
; ENTER Locals, LexLevel

                push    bp          ;Save dynamic link.
                mov     tempreg, sp ;Save for later.
                cmp     LexLevel, 0 ;Done if this is lex level zero.
                je      Lex0
lp:             dec     LexLevel
                jz      Done        ;Quit if at last lex level.
                sub     bp, 2       ;Index into display in prev act rec
                push    [bp]        ; and push each element there.
                jmp     lp          ;Repeat for each entry.

Done:           push    tempreg     ;Add entry for current lex level.
Lex0:           mov     bp, tempreg ;Ptr to current act rec.
                sub     sp, Locals  ;Allocate local storage
```

As you can see from this code, the enter instruction copies the display from activation record to activation record. This can get quite expensive if you nest the procedures to any depth. Most HLLs, if they use the enter instruction at all, always specify a nesting level of zero to avoid copying the display throughout the stack.

The enter instruction puts the value for the display[n] entry at location BP-(n*2). *The enter instruction does not copy the value for display[0] into each stack frame.* Intel assumes that you will keep the main program's global variables in the data segment. To save time and memory, they do not bother copying the display[0] entry.

The enter instruction is very slow, particularly on 80486 and later processors. If you really want to copy the display from activation record to activation record it is probably a better idea to push the items yourself. The following code snippets show how to do this:

```
; enter n, 0    ;14 cycles on the 486

                push    bp          ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; enter n, 1    ;17 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 2       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; enter n, 2    ;20 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                push    [bp-4]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 4       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; enter n, 3    ;23 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                push    [bp-4]      ;4 cycles on the 486
                push    [bp-6]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 6       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486
```

```
; enter n, 4     ;26 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                push    [bp-4]      ;4 cycles on the 486
                push    [bp-6]      ;4 cycles on the 486
                push    [bp-8]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 8       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; etc.
```

If you are willing to believe Intel's cycle timings, you can see that the enter instruction is almost *never* faster than a straight line sequence of instructions that accomplish the same thing. If you are interested in saving space rather than writing fast code, the enter instruction is generally a better alternative. The same is generally true for the leave instruction as well. It is only one byte long, but it is slower than the corresponding mov bp,sp and pop bp instructions.

Accessing non-local variables using the displays created by enter appears in the exercises.

12.2 Passing Variables at Different Lex Levels as Parameters.

Accessing variables at different lex levels in a block structured program introduces several complexities to a program. The previous section introduced you to the complexity of non-local variable access. This problem gets even worse when you try to pass such variables as parameters to another program unit. The following subsections discuss strategies for each of the major parameter passing mechanisms.

For the purposes of discussion, the following sections will assume that "local" refers to variables in the current activation record, "global" refers to variables in the data segment, and "intermediate" refers to variables in some activation record other than the current activation record. Note that the following sections will not assume that ds is equal to ss. These sections will also pass all parameters on the stack. You can easily modify the details to pass these parameters elsewhere.

12.2.1 Passing Parameters by Value in a Block Structured Language

Passing value parameters to a program unit is no more difficult than accessing the corresponding variables; all you need do is push the value on the stack before calling the associated procedure.

To pass a global variable by value to another procedure, you could use code like the following:

```
                push    GlobalVar   ;Assume "GlobalVar" is in DSEG.
                call    Procedure
```

To pass a local variable by value to another procedure, you could use the following code[6]:

```
                push    [bp-2]      ;Local variable in current activation
                call    Procedure   ; record.
```

To pass an intermediate variable as a value parameter, you must first locate that intermediate variable's activation record and then push its value onto the stack. The exact mechanism you use depends on whether you are using static links or a display to keep track of the intermediate variable's activation records. If using static links, you might use

6. The non-global examples all assume the variable is at offset -2 in their activation record. Change this as appropriate in your code.

code like the following to pass a variable from two lex levels up from the current procedure:

```
mov     bx, [bp+4]          ;Assume S.L. is at offset 4.
mov     bx, ss:[bx+4]       ;Traverse two static links
push    ss:[bx-2]           ;Push variables value.
call    Procedure
```

Passing an intermediate variable by value when you are using a display is somewhat easier. You could use code like the following to pass an intermediate variable from lex level one:

```
mov     bx, Display[1*2]    ;Get Display[1] entry.
push    ss:[bx-2]           ;Push the variable's value.
call    Procedure
```

12.2.2 Passing Parameters by Reference, Result, and Value-Result in a Block Structured Language

The pass by reference, result, and value-result parameter mechanisms generally pass the address of parameter on the stack[7]. If global variables reside in the data segment, activation records all exist in the stack segment, and ds≠ss, then you must pass far pointers to access all possible variables[8].

To pass a far pointer you must push a segment value followed by an offset value on the stack. For global variables, the segment value is found in the ds register; for non-global values, ss contains the segment value. To compute the offset portion of the address you would normally use the lea instruction. The following code sequence passes a global variable by reference:

```
push    ds                  ;Push segment adrs first.
lea     ax, GlobalVar       ;Compute offset.
push    ax                  ;Push offset of GlobalVar
call    Procedure
```

Global variables are a special case because the assembler can compute their run-time offsets at assembly time. Therefore, *for scalar global variables only,* we can shorten the code sequence above to

```
push    ds                  ;Push segment adrs.
push    offset GlobalVar    ;Push offset portion.
call    Procedure
```

To pass a local variable by reference you code must first push ss's value onto the stack and then push the local variable's offset. *This offset is the variable's offset within the stack segment, not the offset within the activation record!* The following code passes the address of a local variable by reference:

```
push    ss                  ;Push segment address.
lea     ax, [bp-2]          ;Compute offset of local
push    ax                  ; variable and push it.
call    Procedure
```

To pass an intermediate variable by reference you must first locate the activation record containing the variable so you can compute the effective address into the stack segment. When using static links, the code to pass the parameter's address might look like the following:

7. As you may recall, pass by reference, value-result, and result all use the same calling sequence. The differences lie in the procedures themselves.

8. You can use near pointers if ds=ss or if you keep global variables in the main program's activation record in the stack segment.

```
push    ss                      ;Push segment portion.
mov     bx, [bp+4]              ;Assume S.L. is at offset 4.
mov     bx, ss:[bx+4]           ;Traverse two static links
lea     ax, [bx-2]              ;Compute effective address
push    ax                      ;Push offset portion.
call    Procedure
```

When using a display, the calling sequence might look like the following:

```
push    ss                      ;Push segment portion.
mov     bx, Display[1*2]        ;Get Display[1] entry.
lea     ax, [bx-2]              ;Get the variable's offset
push    ax                      ; and push it.
call    Procedure
```

As you may recall from the previous chapter, there is a second way to pass a parameter by value-result. You can push the value onto the stack and then, when the procedure returns, pop this value off the stack and store it back into the variable from whence it came. This is just a special case of the pass by value mechanism described in the previous section.

12.2.3 Passing Parameters by Name and Lazy-Evaluation in a Block Structured Language

Since you pass the address of a thunk when passing parameters by name or by lazy-evaluation, the presence of global, intermediate, and local variables does not affect the calling sequence to the procedure. Instead, the thunk has to deal with the differing locations of these variables. The following examples will present thunks for pass by name, you can easily modify these thunks for lazy-evaluation parameters.

The biggest problem a thunk has is locating the activation record containing the variable whose address it returns. In the last chapter, this wasn't too much of a problem since variables existed either in the current activation record or in the global data space. In the presence of intermediate variables, this task becomes somewhat more complex. The easiest solution is to pass two pointers when passing a variable by name. The first pointer should be the address of the thunk, the second pointer should be the offset of the activation record containing the variable the thunk must access[9]. When the procedure calls the thunk, it must pass this activation record offset as a parameter to the thunk. Consider the following Panacea procedures:

```
TestThunk:procedure(name item:integer; var j:integer);
begin TestThunk;

        for j in 0..9 do item := 0;

end TestThunk;

CallThunk:procedure;
var
        A: array[0..9] : integer;
        I: integer;
endvar;
begin CallThunk;

        TestThunk(A[I], I);

end CallThunk;
```

The assembly code for the above might look like the following:

```
; TestThunk AR:
;
;       BP+10-    Address of thunk
```

9. Actually, you may need to pass several pointers to activation records. For example, if you pass the variable "A[i,j,k]" by name and A, i, j, and k are all in different activation records, you will need to pass pointers to each activation record. We will ignore this problem here.

```
;       BP+8-     Ptr to AR for Item and J parameters (must be in the same AR).
;       BP+4-     Far ptr to J.

TestThunk       proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    es

                les     bx, [bp+4]              ;Get ptr to J.
                mov     word ptr es:[bx], 0     ;J := 0;
ForLoop:        cmp     word ptr es:[bx], 9     ;Is J > 9?
                ja      ForDone
                push    [bp+8]                  ;Push AR passed by caller.
                call    word ptr [bp+10]        ;Call the thunk.
                mov     word ptr ss:[bx], 0     ;Thunk returns adrs in BX.
                les     bx, [bp+4]              ;Get ptr to J.
                inc     word ptr es:[bx]        ;Add one to it.
                jmp     ForLoop

ForDone:        pop     es
                pop     bx
                pop     ax
                pop     bp
                ret     8
TestThunk       endp


CallThunk       proc    near
                push    bp
                mov     bp, sp
                sub     sp, 12          ;Make room for locals.

                jmp     OverThunk
Thunk           proc
                push    bp
                mov     bp, sp
                mov     bp, [bp+4]      ;Get AR address.
                mov     ax, [bp-22]     ;Get I's value.
                add     ax, ax          ;Double, since A is a word array.
                add     bx, -20         ;Offset to start of A
                add     bx, ax          ;Compute address of A[I] and
                pop     bp              ; return it in BX.
                ret     2               ;Remove parameter from stack.
Thunk           endp

OverThunk:      push    offset Thunk ;Push (near) address of thunk
                push    bp           ;Push ptr to A/I's AR for thunk
                push    ss           ;Push address of I onto stack.
                lea     ax, [bp-22]  ; Offset portion of I.
                push    ax
                call    TestThunk
                mov     sp, bp
                ret
CallThunk       endp
```

12.3 Passing Parameters as Parameters to Another Procedure

When a procedure passes one of its own parameters as a parameter to another procedure, certain problems develop that do not exist when passing variables as parameters. Indeed, in some (rare) cases it is not logically possible to pass some parameter types to some other procedure. This section deals with the problems of passing one procedure's parameters to another procedure.

Pass by value parameters are essentially no different than local variables. All the techniques in the previous sections apply to pass by value parameters. The following sections

deal with the cases where the calling procedure is passing a parameter passed to it by reference, value-result, result, name, and lazy evaluation.

12.3.1 Passing Reference Parameters to Other Procedures

Passing a reference parameter though to another procedure is where the complexity begins. Consider the following (pseudo) Pascal procedure skeleton:

```
procedure HasRef(var refparm:integer);

        procedure ToProc(???? parm:integer);
        begin
            .
            .
            .
        end;

begin {HasRef}
            .
            .
            .
        ToProc(refParm);
            .
            .
            .
end;
```

The "????" in the ToProc parameter list indicates that we will fill in the appropriate parameter passing mechanism as the discussion warrants.

If ToProc expects a pass by value parameter (i.e., ???? is just an empty string), then HasRef needs to fetch the value of the refparm parameter and pass this value to ToProc. The following code accomplishes this[10]:

```
                les     bx, [bp+4]   ;Fetch address of refparm
                push    es:[bx]      ;Push integer pointed at by refparm
                call    ToProc
```

To pass a reference parameter by reference, value-result, or result parameter is easy – just copy the caller's parameter as-is onto the stack. That is, if the parm parameter in ToProc above is a reference parameter, a value-result parameter, or a result parameter, you would use the following calling sequence:

```
                push    [bp+6]       ;Push segment portion of ref parm.
                push    [bp+4]       ;Push offset portion of ref parm.
                call    ToProc
```

To pass a reference parameter by name is fairly easy. Just write a thunk that grabs the reference parameter's address and returns this value. In the example above, the call to ToProc might look like the following:

```
                jmp     SkipThunk
Thunk0          proc    near
                les     bx, [bp+4]   ;Assume BP points at HasRef's AR.
                ret
Thunk0          endp

SkipThunk:      push    offset Thunk0          ;Address of thunk.
                push    bp                     ;AR containing thunk's vars.
                call    ToProc
```

Inside ToProc, a reference to the parameter might look like the following:

```
                push    bp                     ;Save our AR ptr.
                mov     bp, [bp+4]             ;Ptr to Parm's AR.
                call    near ptr [bp+6]        ;Call the thunk.
                pop     bp                     ;Retrieve our AR ptr.
                mov     ax, es:[bx]            ;Access variable.
                  .
                  .
                  .
```

10. The examples in this section all assume the use of a display. If you are using static links, be sure to adjust all the offsets and the code to allow for the static link that the caller must push immediately before a call.

To pass a reference parameter by lazy evaluation is very similar to passing it by name. The only difference (in ToProc's calling sequence) is that the thunk must return the value of the variable rather than its address. You can easily accomplish this with the following thunk:

```
Thunk1          proc    near
                push    es
                push    bx
                les     bx, [bp+4]   ;Assume BP points at HasRef's AR.
                mov     ax, es:[bx]  ;Return value of ref parm in ax.
                pop     bx
                pop     es
                ret
Thunk1          endp
```

12.3.2 Passing Value-Result and Result Parameters as Parameters

Assuming you've created a local variable that holds the value of a value-result or result parameter, passing one of these parameters to another procedure is no different than passing value parameters to other code. Once a procedure makes a local copy of the value-result parameter or allocates storage for a result parameter, you can treat that variable just like a value parameter or a local variable with respect to passing it on to other procedures.

Of course, it doesn't make sense to use the value of a result parameter until you've stored a value into that parameter's local storage. Therefore, take care when passing result parameters to other procedures that you've initialized a result parameter before using its value.

12.3.3 Passing Name Parameters to Other Procedures

Since a pass by name parameter's thunk returns the address of a parameter, passing a name parameter to another procedure is very similar to passing a reference parameter to another procedure. The primary differences occur when passing the parameter on as a name parameter.

When passing a name parameter as a value parameter, you first call the thunk, dereference the address the thunk returns, and then pass the value to the new procedure. The following code demonstrates such a call when the thunk returns the variable's address in es:bx (assume pass by name parameter's AR pointer is at address bp+4 and the pointer to the thunk is at address bp+6):

```
                push    bp                      ;Save our AR ptr.
                mov     bp, [bp+4]              ;Ptr to Parm's AR.
                call    near ptr [bp+6]         ;Call the thunk.
                push    word ptr es:[bx]        ;Push parameter's value.
                pop     bp                      ;Retrieve our AR ptr.
                call    ToProc                  ;Call the procedure.
                  .
                  .
                  .
```

Passing a name parameter to another procedure by reference is very easy. All you have to do is push the address the thunk returns onto the stack. The following code, that is very similar to the code above, accomplishes this:

```
                push    bp                      ;Save our AR ptr.
                mov     bp, [bp+4]              ;Ptr to Parm's AR.
                call    near ptr [bp+6]         ;Call the thunk.
                pop     bp                      ;Retrieve our AR ptr.
                push    es                      ;Push seg portion of adrs.
                push    bx                      ;Push offset portion of adrs.
                call    ToProc                  ;Call the procedure.
                  .
                  .
                  .
```

Passing a name parameter to another procedure as a pass by name parameter is very easy; all you need to do is pass the thunk (and associated pointers) on to the new procedure. The following code accomplishes this:

```
push    [bp+6]                  ;Pass Thunk's address.
push    [bp+4]                  ;Pass adrs of Thunk's AR.
call    ToProc
```

To pass a name parameter to another procedure by lazy evaluation, you need to create a thunk for the lazy-evaluation parameter that calls the pass by name parameter's thunk, dereferences the pointer, and then returns this value. The implementation is left as a programming project.

12.3.4 Passing Lazy Evaluation Parameters as Parameters

Lazy evaluation parameters typically consist of three components: the address of a thunk, a location to hold the value the thunk returns, and a boolean variable that determines whether the procedure must call the thunk to get the parameter's value or if it can simply use the value previously returned by the thunk (see the exercises in the previous chapter to see how to implement lazy evaluation parameters). When passing a parameter by lazy evaluation to another procedure, the calling code must first check the boolean variable to see if the value field is valid. If not, the code must first call the thunk to get this value. If the boolean field is true, the calling code can simply use the data in the value field. In either case, once the value field has data, passing this data on to another procedure is no different than passing a local variable or a value parameter to another procedure.

12.3.5 Parameter Passing Summary

Table 48: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value	Pass the value	Pass address of the value parameter	Pass address of the value parameter	Pass address of the value parameter	Create a thunk that returns the address of the value parameter	Create a thunk that returns the value
Reference	Dereference parameter and pass the value it points at	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Create a thunk that passes the address (value of the reference parameter)	Create a thunk that deferences the reference parameter and returns its value
Value-Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the value-result parameter	Create a thunk that returns the value in the local value of the value-result parameter
Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the result parameter	Create a thunk that returns the value in the local value of the result parameter

Table 48: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Name	Call the thunk, dereference the pointer, and pass the value at the address the thunk returns	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Pass the address of the thunk and any other values associated with the name parameter	Write a thunk that calls the name parameter's thunk, dereferences the address it returns, and then returns the value at that address
Lazy Evaluation	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the local value as the value parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Create a thunk that returns the address of the Lazy Eval's value field	Create a thunk that checks the boolean field of the caller's Lazy Eval parameter. It should call the corresponding thunk if this variable is false. It should set the boolean field to true and then return the data in the value field

12.4 Passing Procedures as Parameters

Many programming languages let you pass a procedure or function name as a parameter. This lets the caller pass along various actions to perform inside a procedure. The classic example is a plot procedure that graphs some generic math function passed as a parameter to plot.

Standard Pascal lets you pass procedures and functions by declaring them as follows:

```
procedure DoCall(procedure x);
begin

       x;

end;
```

The statement DoCall(xyz); calls DoCall that, in turn, calls procedure xyz.

Passing a procedure or function as a parameter may seem like an easy task – just pass the address of the function or procedure as the following example demonstrates:

```
procedure PassMe;
begin
       Writeln('PassMe was called');
end;

procedure CallPassMe(procedure x);
begin
       x;
end;

begin {main}
       CallPassMe(PassMe);
end.
```

The 80x86 code to implement the above could look like the following:

```
PassMe          proc    near
                print
                byte    "PassMe was called",cr,lf,0
                ret
PassMe          endp

CallPassMe      proc    near
                push    bp
                mov     bp, sp
                call    word ptr [bp+4]
                pop     bp
                ret     2
CallPassMe      endp

Main            proc    near
                lea     bx, PassMe   ;Pass address of PassMe to
                push    bx           ; CallPassMe
                call    CallPassMe
                ExitPgm
Main            endp
```

For an example as simple as the one above, this technique works fine. However, it does not always work properly if PassMe needs to access non-local variables. The following Pascal code demonstrates the problem that could occur:

```
program main;

        procedure dummy;
        begin end;

        procedure Recurse1(i:integer; procedure x);

                  procedure Print;
                  begin

                           writeln(i);

                  end;

                  procedure Recurse2(j:integer; procedure y);
                  begin

                           if (j=1) then y
                           else if (j=5) then Recurse1(j-1, Print)
                           else Recurse1(j-1, y);

                  end;

        begin {Recurse1}

                  Recurse2(i, x);

        end;

begin {Main}

        Recurse1(5,dummy);

end.
```

This code produces the following call sequence:

Recurse1(5,dummy) → Recurse2(5,dummy) → Recurse1(4,Print) →
Recurse2(4,Print) → Recurse1(3,Print) → Recurse2(3,Print) →
Recurse1(2,Print) → Recurse2(2,Print) → Recurse1(1,Print) →
Recurse2(1,Print) → Print

Print will print the value of Recurse1's i variable to the standard output. However, there are several activation records present on the stack that raises the obvious question, "which copy of i does Print display?" Without giving it much thought, you might conclude that it should print the value "1" since Recurse2 calls Print when Recurse1's value for i is one. Note, though, that when Recurse2 passes the address of Print to Recurse1, i's value is four. Pascal, like most block structured languages, will use the value of i at the point Recurse2

passes the address of Print to Recurse1. Hence, the code above should print the value four, not the value one.

This creates a difficult implementation problem. After all, Print cannot simply access the display to gain access to the global variable i – the display's entry for Recurse1 points at the latest copy of Recurse1's activation record, not the entry containing the value four which is what you want.

The most common solution in systems using a display is to make a local copy of each display whenever calling a procedure or function. When passing a procedure or function as a parameter, the calling code copies the display along with the address of the procedure or function. This is why Intel's enter instruction makes a copy of the display when building the activation record.

If you are passing procedures and functions as parameters, you may want to consider using static links rather than a display. When using a static link you need only pass a single pointer (the static link) along with the routine's address. Of course, it is more work to access non-local variables, but you don't have to copy the display on every call, which is quite expensive.

The following 80x86 code provides the implementation of the above code using static links:

```
wp              textequ  <word ptr>

Dummy           proc     near
                ret
Dummy           endp

; PrintIt; (Use the name PrintIt to avoid conflict).
;
;       stack:
;
;       bp+4:     static link.

PrintIt         proc     near
                push     bp
                mov      bp, sp
                mov      bx, [bp+4]             ;Get static link
                mov      ax, ss:[bx-10]         ;Get i's value.
                puti
                pop      bp
                ret      2
PrintIt         endp

; Recurse1(i:integer; procedure x);
;
;       stack:
;
;       bp+10:    i
;       bp+8:     x's static link
;       bp+6: x's address

Recurse1        proc     near
                push     bp
                mov      bp, sp
                push     wp [bp+10]             ;Push value of i onto stack.
                push     wp [bp+8]              ;Push x's static link.
                push     wp [bp+6]              ;Push x's address.
                push     bp                     ;Push Recurse1's static link.
                call     Recurse1
                pop      bp
                ret      6
Recurse1        endp

; Recurse2(i:integer; procedure y);
;
;       stack:
;
;       bp+10: j
;       bp+8:  y's static link.
```

```
;       bp+6:  y's address.
;       bp+4:  Recurse2's static link.

Recurse2        proc    near
                push    bp
                mov     bp, sp
                cmp     wp [bp+10], 1           ;Is j=1?
                jne     TryJeq5
                push    [bp+8]                  ;y's static link.
                call    wp [bp+6]               ;Call y.
                jmp     R2Done

TryJeq5:        cmp     wp [bp+10], 5           ;Is j=5?
                jne     Call1
                mov     ax, [bp+10]
                dec     ax
                push    ax
                push    [bp+4]                  ;Push static link to R1.
                lea     ax, PrintIt             ;Push address of print.
                push    ax
                call    Recurse1
                jmp     R2Done

Call1:          mov     ax, [bp+10]
                dec     ax
                push    ax
                push    [bp+8]                  ;Pass along existing
                push    [bp+6]                  ; address and link.
                call    Recurse1

R2Done:         pop     bp
                ret     6
Recurse1        endp

main            proc
                push    bp
                mov     bp, sp
                mov     ax, 5                   ;Push first parameter.
                push    ax
                push    bp                      ;Dummy static link.
                lea     ax, Dummy               ;Push address of dummy.
                push    ax
                call    Recurse1
                pop     bp
                ExitPgm
main            endp
```

There are several ways to improve this code. Of course, this particular program doesn't really need to maintain a display or static list because only PrintIt accesses non-local variables; however, ignore that fact for the time being and pretend it does. Since you know that PrintIt only accesses variables at one particular lex level, and the program only calls PrintIt indirectly, you can pass a pointer to the appropriate activation record; this is what the above code does, although it maintains full static links as well. Compilers must always assume the worst case and often generate inefficient code. If you study your particular needs, however, you may be able to improve the efficiency of your code by avoiding much of the overhead of maintaining static lists or copying displays.

Keep in mind that thunks are special cases of functions that you call indirectly. They suffer from the same problems and drawbacks as procedure and function parameters with respect to accessing non-local variables. If such routines access non-local variables (and thunks almost always will) then you must exercise care when calling such routines. Fortunately, thunks never cause indirect recursion (which is responsible for the crazy problems in the Recurse1 / Recurse2 example) so you can use the display to access any non-local variables appearing within the thunk.

12.5 Iterators

An iterator is a cross between a control structure and a function. Although common high level languages do not often support iterators, they are present in some very high level languages[11]. Iterators provide a combination state machine / function call mechanism that lets a function pick up where it last left off on each new call. Iterators are also part of a loop control structure, with the iterator providing the value of the loop control variable on each iteration.

To understand what an iterator is, consider the following for loop from Pascal:

```
for I := 1 to 10 do <some statement>;
```

When learning Pascal you were probably taught that this statement initializes i with one, compares i with 10, and executes the statement if i is less than or equal to 10. After executing the statement, the for statement increments i and compares it with 10 again, repeating the process over and over again until I is greater than 10.

While this description is semantically correct, and indeed, it's the way that most Pascal compilers implement the for loop, this is not the only point of view that describes how the for loop operates. Suppose, instead, that you were to treat the "to" reserved word as an operator. An operator that expects two parameters (one and ten in this case) and returns the range of values on each successive execution. That is, on the first call the "to" operator would return one, on the second call it would return two, etc. After the tenth call, the "to" operator would *fail* which would terminate the loop. This is exactly the description of an iterator.

In general, an iterator controls a loop. Different languages use different names for iterator controlled loops, this text will just use the name *foreach* as follows:

```
foreach variable in iterator() do
        statements;
endfor;
```

Variable is a variable whose type is compatible with the return type of the iterator. An iterator returns two values: a boolean *success* or *failure* value and a function result. As long as the iterator returns success, the foreach statement assigns the other return value to variable and executes statements. If iterator returns failure, the foreach loop terminates and executes the next sequential statement following the foreach loop's body. In the case of failure, the foreach statement does not affect the value of variable.

Iterators are considerably more complex than normal functions. A typical function call involves two basic operations: a call and a return. Iterator invocations involve four basic operations:

1) Initial iterator call
2) Yielding a value
3) Resumption of an iterator
4) Termination of an iterator.

To understand how an iterator operates, consider the following short example from the Panacea programming language[12]:

```
Range:iterator(start,stop:integer):integer;
begin range;

    while (start <= stop) do

        yield start;
        start := start + 1;

    endwhile;
```

11. Ada and PL/I support very limited forms of iterators, though they do not support the type of iterators found in CLU, SETL, Icon, and other languages.

12. Panacea is a very high level language developed by Randall Hyde for use in compiler courses at UC Riverside.

```
end Range;
```

In the Panacea programming language, iterator calls may only appear in the foreach statement. With the exception of the *yield* statement above, anyone familiar with Pascal or C++ should be able to figure out the basic logic of this iterator.

An iterator in the Panacea programming language may return to its caller using one of two separate mechanisms, it can *return* to the caller, by exiting through the end Range; statement or it may *yield* a value by executing the yield statement. An iterator *succeeds* if it executes the yield statement, it *fails* if it simply returns to the caller. Therefore, the foreach statement will only execute its corresponding statement if you exit an iterator with a yield. The foreach statement terminates if you simply return from the iterator. In the example above, the iterator returns the values start..stop via a yield and then the iterator terminates. The loop

```
foreach i in Range(1,10) do
        write(i);
endfor;
```

is comparable to the Pascal statement:

```
for i := 1 to 10 do write(i);
```

When a Panacea program first executes the foreach statement, it makes an *initial call* to the iterator. The iterator runs until it executes a yield or it returns. If it executes the yield statement, it returns the value of the expression following the yield as the iterator result and it returns success. If it simply returns, the iterator returns failure and no iterator result. In the current example, the initial call to the iterator returns success and the value one.

Assuming a successful return (as in the current example), the foreach statement assigns the iterator return value to the loop control variable and executes the foreach loop body. After executing the loop body, the foreach statement calls the iterator again. However, this time the foreach statement *resumes* the iterator rather than making an initial call. *An iterator resumption continues with the first statement following the last yield it executed.* In the range example, a resumption would continue execution at the start := start + 1; statement. On the first resumption, the Range iterator would add one to start, producing the value two. Two is less than ten (stop's value) so the while loop would repeat and the iterator would yield the value two. This process would repeat over and over again until the iterator yields ten. Upon resuming after yielding ten, the iterator would increment start to eleven and then return, rather than yield, since this new value is not less than or equal to ten. When the range iterator returns (fails), the foreach loop terminates.

12.5.1 Implementing Iterators Using In-Line Expansion

The implementation of an iterator is rather complex. To begin with, consider a first attempt at an assembly implementation of the foreach statement above:

```
                push    1                       ;Assume 286 or better
                push    10                      ; and parms passed on stack.
                call    Range_Initial           ;Make initial call to iter.
                jc      Failure                 ;C=0, 1 means success, fail.
ForLoop:        puti                            ;Assume result is in AX.
                call    Range_Resume            ;Resume iterator.
                jnc     ForLoop                 ;Carry clear is success!
Failure:
```

Although this looks like a straight-forward implementation project, there are several issues to consider. First, the call to Range_Resume above looks simple enough, but there is no fixed address that corresponds to the resume address. While it is certainly true that this Range example has only one resume address, in general you can have as many yield statements as you like in an iterator. For example, the following iterator returns the values 1, 2, 3, and 4:

```
OneToFour:iterator:integer;
begin OneToFour;

          yield 1;
          yield 2;
          yield 3;
          yield 4;

end OneToFour;
```

The initial call would execute the `yield 1;` statement. The first resumption would execute the `yield 2;` statement, the second resumption would execute `yield 3;`, etc. Obviously there is no single resume address the calling code can count on.

There are a couple of additional details left to consider. First, an iterator is free to call procedures and functions[13]. If such a procedure or function executes the `yield` statement then resumption by the `foreach` statement continues execution within the procedure or function that executed the `yield`. Second, the semantics of an iterator require all local variables and parameters to maintain their values until the iterator terminates. That is, yielding does not deallocate local variables and parameters. Likewise, any return addresses left on the stack (e.g., the call to a procedure or function that executes the `yield` statement) must not be lost when a piece of code yields and the corresponding `foreach` statement resumes the iterator. In general, this means you cannot use the standard call and return sequence to yield from or resume to an iterator because you have to preserve the contents of the stack.

While there are several ways to implement iterators in assembly language, perhaps the most practical method is to have the iterator call the loop controlled by the iterator and have the loop return back to the iterator function. Of course, this is counter-intuitive. Normally, one thinks of the iterator as the function that the loop calls on each iteration, not the other way around. However, given the structure of the stack during the execution of an iterator, the counter-intuitive approach turns out to be easier to implement.

Some high level languages support iterators in exactly this fashion. For example, Metaware's Professional Pascal Compiler for the PC supports iterators[14]. Were you to create a code sequence as follows:

```
iterator OneToFour:integer;
begin
          yield 1;
          yield 2;
          yield 3;
          yield 4;
end;
```

and call it in the main program as follows:

```
for i in OneToFour do writeln(i);
```

Professional Pascal would completely rearrange your code. Instead of turning the iterator into an assembly language function and call this function from within the for loop body, this code would turn the `for` loop body into a function, expand the iterator in-line (much like a macro) and call the `for` loop body function on each `yield`. That is, Professional Pascal would probably produce assembly language that looks something like the following:

13. In Panacea an iterator could also call other types of program units, including other iterators, but you can ignore this for now.

14. Obviously, this is a non-standard extension to the Pascal programming language provided in Professional Pascal.

```
; The following procedure corresponds to the for loop body
; with a single parameter (I) corresponding to the loop
; control variable:

ForLoopCode     proc    near
                push    bp
                mov     bp, sp
                mov     ax, [bp+4]  ;Get loop control value and
                puti                ; print it.
                putcr
                pop     bp
                ret     2           ;Pop loop control value off stk.
ForLoopCode     endp

; The follow code would be emitted in-line upon encountering the
; for loop in the main program, it corresponds to an in-line
; expansion of the iterator as though it were a macro,
; substituting a call for the yield instructions:

                push    1           ;On 286 and later processors only.
                call    ForLoopCode
                push    2
                call    ForLoopCode
                push    3
                call    ForLoopCode
                push    4
                call    ForLoopCode
```

This method for implementing iterators is convenient and produces relatively efficient (fast) code. It does, however, suffer from a couple drawbacks. First, since you must expand the iterator in-line wherever you call it, much like a macro, your program could grow large if the iterator is not short and you use it often. Second, this method of implementing the iterator completely hides the underlying logic of the code and makes your assembly language programs difficult to read and understand.

12.5.2 Implementing Iterators with Resume Frames

In-line expansion is not the only way to implement iterators. There is another method that preserves the structure of your program at the expense of a slightly more complex implementation. Several high level languages, including Icon and CLU, use this implementation.

To start with, you will need another stack frame: the *resume frame.* A resume frame contains two entries: a yield return address (that is, the address of the next instruction after the yield statement) and a *dynamic link*, which is a pointer to the iterator's activation record. Typically the dynamic link is just the value in the bp register at the time you execute the yield instruction. This version implements the four parts of an iterator as follows:

1) A call instruction for the initial iterator call,
2) A call instruction for the yield statement,
3) A ret instruction for the resume operation, and
4) A ret instruction to terminate the iterator.

To begin with, an iterator will require *two* return addresses rather than the single return address you would normally expect. The first return address appearing on the stack is the termination return address. The second return address is where the subroutine transfers control on a yield operation. The calling code must push these two return addresses upon initial invocation of the iterator. The stack, upon initial entry into the iterator, should look something like Figure 12.8.

As an example, consider the Range iterator presented earlier. This iterator requires two parameters, a starting value and an ending value:

```
foreach i in Range(1,10) do writeln(i);
```

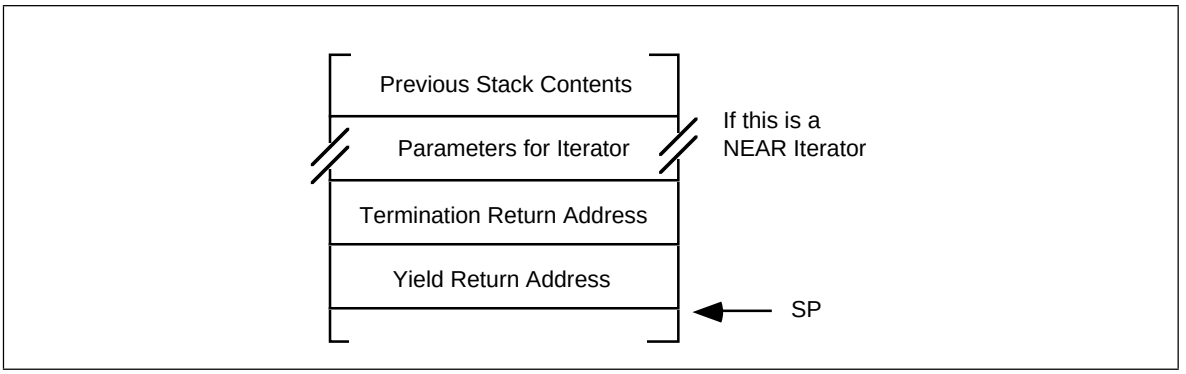


Figure 12.8 Iterator Activation Record

The code to make the initial call to the Range iterator, producing a stack like the one above, could be the following:

```
                push    1                       ;Push start parameter value.
                push    10                      ;Push stop parameter value.
                push    offset ForDone          ;Push termination address.
                call    Range                   ;Pushes yield return address.
```

ForDone is the first statement immediately following the foreach loop, that is, the instruction to execute when the iterator returns failure. The foreach loop body must begin with the first instruction following the call to Range. At the end of the foreach loop, rather than jumping back to the start of the loop, or calling the iterator again, this code should just execute a ret instruction. The reason will become clear in a moment. So the implementation of the above foreach statement could be the following:

```
                push    1                       ;Obviously, this requires a
                push    10                      ; 80286 or later processor.
                push    offset ForDone
                call    Range
                mov     bp, [bp]                ;Explained a little later.
                puti
                putcr
                ret
ForDone:
```

Granted, this doesn't look anything at all like a loop. However, by playing some *major* tricks with the stack, you'll see that this code really does iterate the loop body (puti and putcr) as intended.

Now consider the Range iterator itself, here's the code to do the job:

```
Range_Start     equ     word ptr <[bp+8]>       ;Address of Start parameter.
Range_Stop      equ     word ptr <[bp+6]>       ;Address of Stop parameter.
Range_Yield     equ     word ptr <[bp+2]>       ;Yield return address.

Range           proc    near
                push    bp
                mov     bp, sp
RangeLoop:      mov     ax, Range_Start         ;Get start parameter and
                cmp     ax, Range_Stop          ; compare against stop.
                ja      RangeDone               ;Terminate if start > stop

; Okay, build the resume frame:

                push    bp                      ;Save dynamic link.
                call    Range_Yield             ;Do YIELD operation.
                pop     bp                      ;Restore dynamic link.
                inc     Range_Start             ;Bump up start value
                jmp     RangeLoop               ;Repeat until start > stop.

RangeDone:      pop     bp                      ;Restore old BP
                add     sp, 2                   ;Pop YIELD return address
                ret     4                       ;Terminate iterator.
Range           endp
```

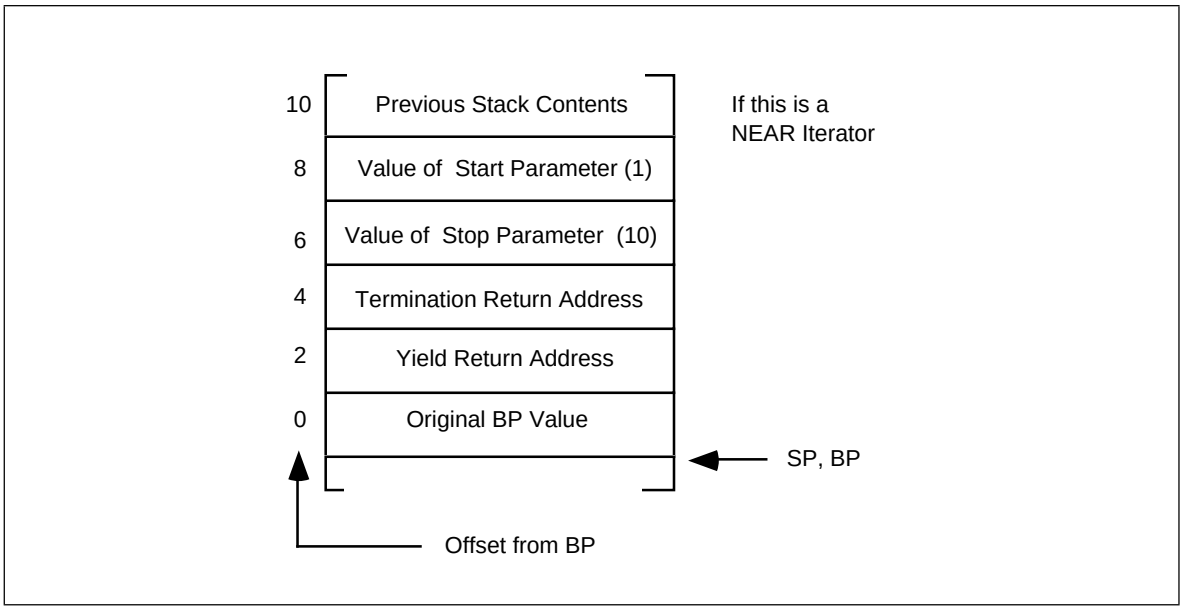


Figure 12.9 Range Activation Record

Although this routine is rather short, don't let its size deceive you; it's quite complex. The best way to describe how this iterator operates is to take it a few instructions at a time. The first two instructions are the standard entry sequence for a procedure. Upon execution of these two instructions, the stack looks like that in Figure 12.9.

The next three statements in the Range iterator, at label RangeLoop, implement the termination test of the while loop. When the Start parameter contains a value greater than the Stop parameter, control transfers to the RangeDone label at which point the code pops the value of bp off the stack, pops the yield return address off the stack (since this code will *not* return back to the body of the iterator loop) and then returns via the termination return address that is immediately above the yield return address on the stack. The return instruction also pops the two parameters off the stack.

The real work of the iterator occurs in the body of the while loop. The push, call, and pop instructions implement the yield statement. The push and call instructions build the resume frame and then return control to the body of the foreach loop. The call instruction *is not* calling a subroutine. What it is really doing here is finishing off the resume frame (by storing the yield resume address into the resume frame) and then it *returns* control back to the body of the foreach loop by jumping indirect through the yield return address pushed on the stack by the initial call to the iterator. After the execution of this call, the stack frame looks like that in Figure 12.9.

Also note that the ax register contains the return value for the iterator. As with functions, ax is a good place to return the iterator return result.

Immediately after yielding back to the foreach loop, the code must reload bp with the original value prior to the iterator invocation. This allows the calling code to correctly access parameters and local variables in its own activation record rather than the activation record of the iterator. Since bp just happens to point at the original bp value for the calling code, executing the mov bp, [bp] instruction reloads bp as appropriate. Of course, in this example reloading bp isn't necessary because the body of the foreach loop does not reference any memory locations off the bp register, but in general you will need to restore bp.

At the end of the foreach loop body the ret instruction resumes the iterator. The ret instruction pops the return address off the stack which returns control back to the iterator immediately after the call. The instruction at this point pops bp off the stack, increments the Start variable, and then repeats the while loop.

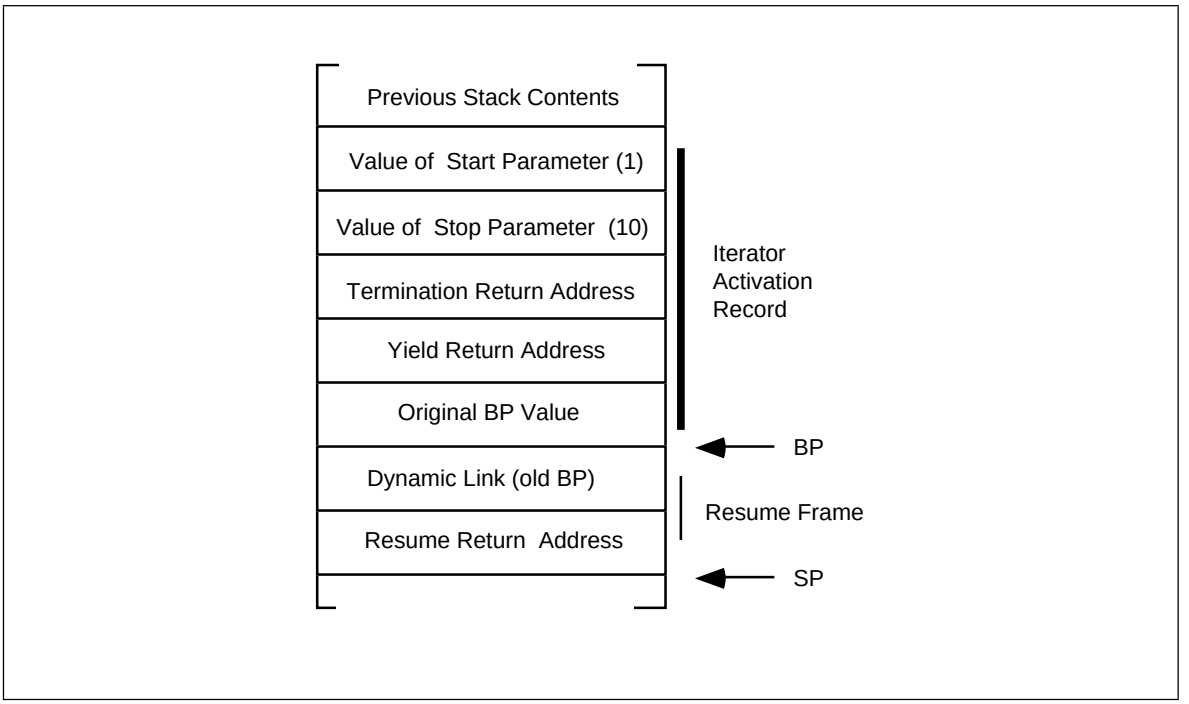


Figure 12.10 Range Resume Record

Of course, this is a lot of work to create a piece of code that simply repeats a loop ten times. A simple *for* loop would have been much easier and quite a bit more efficient that the foreach implementation described in this section. This section used the Range iterator because it was easy to show how iterators work using Range, not because actually implementing Range as an iterator is a good idea.

12.6 Sample Programs

The sample programs in this chapter provide two examples of iterators. The first example is a simple iterator that processes characters in a string and returns the vowels found in that string. The second iterator is a synthetic program (i.e., written just to demonstrate iterators) that is considerably more complex since it deals with static links. The second sample program also demonstrates another way to build the resume frame for an iterator. Take a good look at the macros that this program uses. They can simplify the user of iterators in your programs.

12.6.1 An Example of an Iterator

The following example demonstrates a simple iterator. This piece of code reads a string from the user and then locates all the vowels (a, e, i, o, u, w, y) on the line and prints their index into the string, the vowel at that position, and counts the occurrences of each vowel. This isn't a particularly good example of an iterator, however it does serve to demonstrate an implementation and use.

First, a pseudo-Pascal version of the program:

```
program DoVowels(input,output);
const
       Vowels = [‘a’, ‘e’, ‘i’, ‘o’, ‘u’, ‘y’, ‘w’,
                 ‘A’, ‘E’, ‘I’, ‘O’, ‘U’, ‘Y’, ‘W’];

var
```

```
        ThisVowel : integer;
        VowelCnt : array [char] of integer;

        iterator GetVowel(s:string) : integer;
        var
                CurIndex : integer;
        begin

                for CurIndex := 1 to length(s) do
                        if (s [CurIndex] in Vowels) then begin

                           { If we have a vowel, bump the cnt by 1 }
                           Vowels[s[CurIndex]] := Vowels[s[CurIndex]]+1;

                           ( Return index into string of current vowel }
                           yield CurIndex;

                        end;
        end;

begin {main}

        { First, initialize our vowel counters }

        VowelCnt [‘a’] := 0;
        VowelCnt [‘e’] := 0;
        VowelCnt [‘i’] := 0;
        VowelCnt [‘o’] := 0;
        VowelCnt [‘u’] := 0;
        VowelCnt [‘w’] := 0;
        VowelCnt [‘y’] := 0;
        VowelCnt [‘A’] := 0;
        VowelCnt [‘E’] := 0;
        VowelCnt [‘I’] := 0;
        VowelCnt [‘O’] := 0;
        VowelCnt [‘U’] := 0;
        VowelCnt [‘W’] := 0;
        VowelCnt [‘Y’] := 0;

        { Read and process the input string}

        Write(‘Enter a string: ‘);
        ReadLn(InputStr);
        foreach ThisVowel in GetVowel(InputStr) do
                WriteLn(‘Vowel ‘,InputStr [ThisVowel],
                        ‘ at position ‘, ThisVowel);

        { Output the vowel counts }

        WriteLn(‘# of A’’s:’,VowelCnt[‘a’] + VowelCnt[‘A’]);
        WriteLn(‘# of E’’s:’,VowelCnt[‘e’] + VowelCnt[‘E’]);
        WriteLn(‘# of I’’s:’,VowelCnt[‘i’] + VowelCnt[‘I’]);
        WriteLn(‘# of O’’s:’,VowelCnt[‘o’] + VowelCnt[‘O’]);
        WriteLn(‘# of U’’s:’,VowelCnt[‘u’] + VowelCnt[‘U’]);
        WriteLn(‘# of W’’s:’,VowelCnt[‘w’] + VowelCnt[‘W’]);
        WriteLn(‘# of Y’’s:’,VowelCnt[‘y’] + VowelCnt[‘Y’]);

end.
```

Here’s the working assembly language version:

```
                .286        ;For PUSH imm instr.
                .xlist
                include  stdlib.a
                includelib stdlib.lib
                .list

; Some “cute” equates:

Iterator        textequ  <proc>
endi            textequ  <endp>
wp              textequ  <word ptr>

; Necessary global variables:

dseg            segment  para public ‘data’
```

```
; As per UCR StdLib instructions, InputStr must hold
; at least 128 characters.

InputStr        byte    128 dup (?)

; Note that the following statement initializes the
; VowelCnt array to zeros, saving us from having to
; do this in the main program.

VowelCnt        word    256 dup (0)

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; GetVowel-     This iterator searches for the next vowel in the
;               input string and returns the index to the value
;               as the iterator result. On entry, ES:DI points
;               at the string to process. On yield, AX returns
;               the zero-based index into the string of the
;               current vowel.
;
; GVYield-      Address to call when performing the yield.
; GVStrPtr-     A local variable that points at our string.

GVYield         texequ  <word ptr [bp+2]>
GVStrPtr        texequ  <dword ptr [bp-4]>

GetVowel        Iterator
                push    bp
                mov     bp, sp

; Create and initialize GVStrPtr. This is a pointer to the
; next character to process in the input string.

                push    es
                push    di

; Save original ES:DI values so we can restore them on YIELD
; and on termination.

                push    es
                push    di

; Okay, here's the main body of the iterator. Fetch each
; character until the end of the string and see if it is
; a vowel. If it is a vowel, yield the index to it. If
; it is not a vowel, move on to the next character.

GVLoop:         les     di, GVStrPtr ;Ptr to next char.
                mov     al, es:[di]  ;Get this character.
                cmp     al, 0        ;End of string?
                je      GVDone

; The following statement will convert all lower case
; characters to upper case. It will also translate other
; characters to who knows what, but we don't care since
; we only look at A, E, I, O, U, W, and Y.

                and     al, 5fh

; See if this character is a vowel. This is a disgusting
; set membership operation.

                cmp     al, 'A'
                je      IsAVowel
                cmp     al, 'E'
                je      IsAVowel
                cmp     al, 'I'
                je      IsAVowel
                cmp     al, 'O'
                je      IsAVowel
                cmp     al, 'U'
                je      IsAVowel
                cmp     al, 'W'
                je      IsAVowel
```

```
                cmp     al, 'Y'
                jne     NotAVowel

; If we've got a vowel we need to yield the index into
; the string to that vowel. To compute the index, we
; restore the original ES:DI values (which points at
; the beginning of the string) and subtract the current
; position (now in AX) from the first position. This
; produces a zero-based index into the string.
; This code must also increment the corresponding entry
; in the VowelCnt array so we can print the results
; later. Unlike the Pascal code, we've converted lower
; case to upper case so the count for upper and lower
; case characters will appear in the upper case slot.

IsAVowel:       push    bx          ;Bump the vowel
                mov     ah, 0       ; count by one.
                mov     bx, ax
                shl     bx, 1
                inc     VowelCnt[bx]
                pop     bx

                mov     ax, di
                pop     di          ;Restore original DI
                sub     ax, di      ;Compute index.
                pop     es          ;Restore original ES

                push    bp          ;Save our frame pointer
                call    GVYield     ;Yield to caller
                pop     bp          ;Restore our frame pointer
                push    es          ;Save ES:DI again
                push    di

; Whether it was a vowel or not, we've now got to move
; on to the next character in the string. Increment
; our string pointer by one and repeat the process
; over again.

NotAVowel:      inc     wp GVStrPtr
                jmp     GVLoop

; If we've reached the end of the string, terminate
; the iterator here. We need to restore the original
; ES:DI values, remove local variables, pop the YIELD
; address, and then return to the termination address.

GVDone:         pop     di          ;Restore ES:DI
                pop     es
                mov     sp, bp      ;Remove locals
                add     sp, 2       ;Pop YIELD address
                pop     bp
                ret
GetVowel        endi

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                print
                byte    "Enter a string: ",0
                lesi    InputStr
                gets                ;Read input line.

; The following is the foreach loop. Note that the label
; "FOREACH" is present for documentation purpose only.
; In fact, the foreach loop always begins with the first
; instruction after the call to GetVowel.
;
; One other note: this assembly language code uses
; zero-based indexes for the string. The Pascal version
; uses one-based indexes for strings. So the actual
; numbers printed will be different. If you want the
; values printed by both programs to be identical,
```

```
; uncomment the INC instruction below.

                push    offset ForDone          ;Termination address.
                call    GetVowel                ;Start iterator
FOREACH:        mov     bx, ax
                print
                byte    "Vowel ",0
                mov     al, InputStr[bx]
                putc
                print
                byte    " at position ",0
                mov     ax, bx
;               inc     ax
                puti
                putcr
                ret                             ;Iterator resume.

ForDone:        printf
                byte    "# of A's: %d\n"
                byte    "# of E's: %d\n"
                byte    "# of I's: %d\n"
                byte    "# of O's: %d\n"
                byte    "# of U's: %d\n"
                byte    "# of W's: %d\n"
                byte    "# of Y's: %d\n",0
                dword   VowelCnt + ('A'*2)
                dword   VowelCnt + ('E'*2)
                dword   VowelCnt + ('I'*2)
                dword   VowelCnt + ('O'*2)
                dword   VowelCnt + ('U'*2)
                dword   VowelCnt + ('W'*2)
                dword   VowelCnt + ('Y'*2)

Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.6.2 Another Iterator Example

One problem with the iterator examples appearing in this chapter up to this point is that they do not access any global or intermediate variables. Furthermore, these examples do not work if an iterator is recursive or calls other procedures that yield the value to the foreach loop. The major problem with the examples up to this point has been that the foreach loop body has been responsible for reloading the bp register with a pointer to the foreach loop's procedure's activation record. Unfortunately, the foreach loop body has to assume that bp currently points at the iterator's activation record so it can get a pointer to its own activation record from that activation record. This will not be the case if the iterator's activation record is not the one on the top of the stack.

To rectify this problem, the code doing the yield operation must set up the bp register so that it points at the activation record of the procedure containing the foreach loop before returning back to the loop. This is a somewhat complex operation. The following macro accomplishes this from inside an iterator:

```
Yield           macro
                mov     dx, [BP+2]              ;Place to yield back to.
                push    bp                      ;Save Iterator link
                mov     bp, [bp]                ;Get ptr to caller's A.R.
```

```
                call    dx                      ;Push resume address and rtn.
                pop     bp                      ;Restore ptr to our A. R.
                endm
```

Note an unfortunate side effect of this code is that it modifies the dx register. Therefore, the iterator does not preserve the dx register across a call to the iterator function.

The macro above assumes that the bp register points at the iterator's activation record. If it does not, then you must execution some additional instructions to follow the static links back to the iterator's activation record to obtain the address of the foreach loop procedure's activation record.

```
; ITERS.ASM
;
; Roughly corresponds to the example in Ghezzi & Jazayeri's
; "Programming Language Concepts" text.
;
; Randall Hyde
;
;
; This program demonstrates an implementation of:
;
;  l := 0;
;  foreach i in range(1,3) do
;      foreach j in iter2() do
;                writeln(i, ',', j, ',', l):
;
;
;  iterator range(start,stop):integer;
;  begin
;
;      while start <= stop do begin
;
;                yield start;
;                start := start+1;
;      end;
;  end;
;
;  iterator iter2:integer;
;  var k:integer;
;  begin
;
;      foreach k in iter3 do
;                yield k;
;  end;
;
;  iterator iter3:integer;
;  begin
;
;      l := l + 1;
;      yield 1;
;      l := l + 1;
;      yield 2;
;      l := l + 1;
;      yield 0;
;  end;
;
;
; This code will print:
;
;      1, 1, 1
;      1, 2, 2
;      1, 0, 3
;      2, 1, 4
;      2, 2, 5
;      2, 0, 6
;      3, 1, 7
;      3, 2, 8
;      3, 0, 9
```

```
                .xlist
                include  stdlib.a
                includelibstdlib.lib
                .list

                .286                            ;Allow extra adrs modes.

dseg            segment  para stack 'data'

; Put the stack in the data segment so we can use the small memory model
; to simplify addressing:

stk             byte     1024 dup ('stack')
EndStk          word     0

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg, ss:dseg

; Here's the structure of a resume frame.  Note that this structure isn't
; actually used in this code.  It is only provided to show you what data
; is sitting on the stack when Yield builds a resume frame.

RsmFrm          struct
ResumeAdrs      word     ?
IteratorLink    word     ?
RsmFrm          ends


; The following macro builds a resume frame and the returns to the caller
; of an iterator.  It assumes that the iterator and whoever called the
; iterator have the standard activation record defined above and that we
; are building the standard resume frame described above.
;
; This code wipes out the DX register.  Whoever calls the iterator cannot
; count on DX being preserved, likewise, the iterator cannot count on DX
; being preserved across a yield.  Presumably, the iterator returns its
; value in AX.

ActRec          struct
DynamicLink     word     ?           ;Saved BP value.
YieldAdrs       word     ?           ;Return Adrs for proc.
StaticLink      word     ?           ;Static link for proc.
ActRec          ends

AR              equ      [bp].ActRec

Yield           macro
                mov      dx, AR.YieldAdrs       ;Place to yield back to.
                push     bp                     ;Save Iterator link
                mov      bp, AR.DynamicLink     ;Get ptr to caller's A.R.
                call     dx                     ;Push resume address and rtn.
                pop      bp                     ;Restore ptr to our A. R.
                endm


; Range(start, stop) - Yields start..stop and then fails.

; The following structure defines the activation record for Range:

rngAR           struct
DynamicLink     word     ?           ;Saved BP value.
YieldAdrs       word     ?           ;Return Adrs for proc.
StaticLink      word     ?           ;Static link for proc.
FailAdrs        word     ?           ;Go here when we fail
Stop            word     ?           ;Stop parameter
Start           word     ?           ;Start parameter
```

```
rngAR           ends

rAR             equ     [bp].rngAR


Range           proc
                push    bp
                mov     bp, sp

; While start <= stop, yield start:

WhlStartLEStop: mov     ax, rAR.Start ;Also puts return value
                cmp     ax, rAR.Stop  ; in AX.
                jnle    RangeFail

                yield

                inc     rAR.Start
                jmp     WhlStartLEStop

RangeFail:      pop     bp            ;Restore Dynamic Link.
                add     sp, 4         ;Skip ret adrs and S.L.
                ret     4             ;Return through fail address.
Range           endp


; Iter2- Just calls iter3() and returns whatever value it generates.
;
; Note: Since iter2 and iter3 are at the same lex level, the static link
; passed to iter3 must be the same as the static link passed to iter2.
; This is why the "push [bp]" instruction appears below (as opposed to the
; "push bp" instruction which appears in the calls to Range and iter2).
; Keep in mind, Range and iter2 are only called from main and bp contains
; the static link at that point.  This is not true when iter2 calls iter3.

iter2           proc
                push    bp
                mov     bp, sp

                push    offset i3Fail ;Failure address.
                push    [bp]          ;Static link is link to main.
                call    iter3
                yield                 ;Return value returned by iter3
                ret                   ;Resume Iter3.

i3Fail:         pop     bp            ;Restore Dynamic Link.
                add     sp, 4         ;Skip return address & S.L.
                ret                   ;Return through fail address.
iter2           endp


; Iter3() simply yields the values 1, 2, and 0:

iter3           proc
                push    bp
                mov     bp, sp

                mov     bx, AR.StaticLink;Point BX at main's AR.
                inc     word ptr [bx-6];Increment L in main.
                mov     ax, 1
                yield

                mov     bx, AR.StaticLink
                inc     word ptr [bx-6]
                mov     ax, 2
                yield
                mov     bx, AR.StaticLink
                inc     word ptr [bx-6]
                mov     ax, 0
                yield
```

```
                pop     bp          ;Restore Dynamic Link.
                add     sp, 4       ;Skip return address & S.L.
                ret                 ;Return through fail address.
iter3           endp


; Main's local variables are allocated on the stack in order to justify
; the use of static links.

i               equ     [bp-2]
j               equ     [bp-4]
l               equ     [bp-6]

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                mov     ss, ax
                mov     sp, offset EndStk

; Allocate storage for i, j, and l on the stack:

                mov     bp, sp
                sub     sp, 6

                meminit

                mov     word ptr l, 0 ;Initialize l.

; foreach i in range(1,3) do:

                push    1             ;Parameters.
                push    3
                push    offset iFail  ;Failure address.
                push    bp           ;Static link points at our AR.
                call    Range

; Yield from range comes here.  The label is for your benefit.

RangeYield:     mov     i, ax       ;Save away loop control value.

; foreach j in iter2 do:

                push    offset jfail  ;Failure address.
                push    bp            ;Static link points at our AR.
                call    iter2

; Yield from iter2 comes here:

iter2Yield:     mov     j, ax

                mov     ax, i
                puti
                print
                byte    ", ",0
                mov     ax, j
                puti
                print
                byte    ", ",0
                mov     ax, l
                puti
                putcr

; Restart iter2:

                ret                 ;Resume iterator.

; Restart Range down here:
```

```
jFail:          ret                     ;Resume iterator.

; All Done!

iFail:          print
                byte    cr,lf,"All Done!",cr,lf,0

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

; zzzzzzseg must be the last segment that gets loaded into memory!
; This is where the heap begins.

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.7 Laboratory Exercises

This chapter's laboratory exercises consist of three components. In the first exercise you will experiment with a fairly complex set of iterators. In the second exercise you will learn how the 80286's enter and leave instructions operate. In the third exercise, you will run some experiments on parameter passing mechanisms.

12.7.1 Iterator Exercise

In this laboratory exercise you will be working with a program (Ex12_1.asm on the companion CD-ROM) that uses four iterators. The first three iterators perform some fairly simple computations, the fourth iterator returns (successively) pointers to the first three iterators' code that the main program can use to call these iterators.

For your lab report: study the following code and explain how it works. Run it and explain the output. Assemble the program with the "/Zi" option, then from within CodeView, set a breakpoint on the first instruction of the four iterators. Run the program up to these break points and dump the memory starting at the current stack pointer value (ss:sp). Describe the meaning of the data on the stack at each breakpoint. Also, set a breakpoint on the "call ax" instruction. Trace into the routine ax points at upon each breakpoint and describe which routine this instruction calls. How many times does this instruction execute?

```
; EX12_1.asm
;
; Program to support the laboratory exercise in Chapter 12.
;
; This program combines iterators, passing parameters as parameters,
; and procedural parameters all into the same program.
;
;
; This program implements the following iterators (examples written in panacea):
;
; program EX12_1;
;
; fib:iterator(n:integer):integer;
; var
;      CurIndex:integer;
;      Fn1:       integer;
;      Fn2:       integer;
; endvar;
; begin fib;
;
```

```
;       yield 1;  (* Always have at least n=0 *)
;       if (n <> 0) then
;
;                 yield 1; (* Have at least n=1 at this point *)
;
;                 Fn1 := 1;
;                 Fn2 := 1;
;                 foreach CurIndex in 2..n do
;
;                           yield Fn1+Fn2;
;                           Fn2 = Fn1;
;                           Fn1 = CurIndex;
;
;                 endfor;
;       endif;
;
; end fib;
;
;
;
; UpDown:iterator(n:integer):integer;
; var
;       CurIndex:integer;
; endvar;
; begin UpDown;
;
;       foreach CurIndex in 0..n do
;
;                 yield CurIndex;
;
;       endfor;
;       foreach CurIndex in n-1..0 do
;
;                 yield CurIndex;
;
;       endfor;
;
; end UpDown;
;
;
;
; SumToN:iterator(n:integer):integer;
; var
;       CurIndex:integer;
;       Sum:       integer;
; endvar;
; begin SumToN;
;
;       Sum := 0;
;       foreach CurIndex in 0..n do
;
;                 Sum := Sum + CurIndex;
;                 yield Sum;
;
;       endfor;
;
; end SumToN;
;
;
; MultiIter returns a pointer to an iterator that accepts a single integer
parameter.
;
; MultiIter: iterator: [iterator(n:integer)];
; begin MultiIter;
;
;       yield @Fib;(* Return pointers to the three iterators above *)
;       yield @UpDown;(* as the result of this iterator.*)
;       yield @SumToN;
;
; end MultiIter;
```

```
;
;
; var
;      i:integer;
;      n:integer;
;      iter:[iterator(n:integer)];
; endvar;
;
; begin EX12_1;
;
;      (* The following for loop repeats six times, passing its loop index as*)
;      (* the parameter to the Fib, UpDown, and SumToN parameters.*)
;
;      foreach n in 0..5 do
;
;
;      (* The following (funny looking) iterator sequences through *)
;      (* each of the three iterators: Fib, UpDown, and SumToN.  It*)
;      (* returns a pointer as the iterator value.  The innermost  *)
;      (* foreach loop uses this pointer to call the appropriate   *)
;      (* iterator.                                                *)
;
;                foreach iter in MultiIter do
;
;                (* Okay, this for loop invokes whatever iterator was *)
;                (* return by the MultiIter iterator above.           *)
;
;                         foreach i in [MultiIter](n) do
;
;                           write(i:3);
;
;                         endfor;
;                         writeln;
;
;                endfor;
;                writeln;
;
;      endfor;
;
; end EX12_1;

                .xlist
                include  stdlib.a
                includelibstdlib.lib
                .list

                .286                            ;Allow extra adrs modes.


wp              textequ  <word ptr>
ofs             textequ  <offset>


dseg            segment  para public 'code'
dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ss:sseg


; The following macro builds a resume frame and the returns to the caller
; of an iterator.  It assumes that the iterator and whoever called the
; iterator have the standard activation record defined above and that we
; are building the standard resume frame described above.
;
; This code wipes out the DX register.  Whoever calls the iterator cannot
; count on DX being preserved, likewise, the iterator cannot count on DX
; being preserved across a yield.  Presumably, the iterator returns its
; value in AX.
```

```
Yield           macro
                mov     dx, [BP+2]   ;Place to yield back to.
                push    bp           ;Save Iterator link
                mov     bp, [bp]     ;Get ptr to caller's A.R.
                call    dx           ;Push resume address and rtn.
                pop     bp           ;Restore ptr to our A. R.
                endm


; Fib(n) - Yields the sequence of fibonacci numbers from F(0)..F(n).
;         The fibonacci sequence is defined as:
;
;               F(0) and F(1) = 1.
;               F(n) = F(n-1) + F(n-2) for n > 1.


; The following structure defines the activation record for Fib

CurIndex        textequ  <[bp-6]>     ;Current sequence value.
Fn1             textequ  <[bp-4]>     ;F(n-1) value.
Fn2             textequ  <[bp-2]>     ;F(n-2) value.
DynamicLink     textequ  <[bp]>       ;Saved BP value.
YieldAdrs       textequ  <[bp+2]>     ;Return Adrs for proc.
FailAdrs        textequ  <[bp+4]>     ;Go here when we fail
n               textequ  <[bp+6]>     ;The initial parameter


Fib             proc
                push    bp
                mov     bp, sp
                sub     sp, 6        ;Make room for local variables.

; We will also begin yielding values starting at F(0).
; Since F(0) and F(1) are special cases, yield their values here.

                mov     ax, 1        ;Yield F(0) (we always return at least
                yield                ; F(0)).

                cmp     wp n, 1      ;See if user called this with n=0.
                jb      FailFib
                mov     ax, 1
                yield

; Okay, n >=1 so we need to go into a loop to handle the remaining values.
; First, begin by initializing Fn1 and Fn2 as appropriate.

                mov     wp Fn1, 1
                mov     wp Fn2, 1
                mov     wp CurIndex, 2

WhlLp:          mov     ax, CurIndex ;See if CurIndex > n.
                cmp     ax, n
                ja      FailFib

                push    Fn1
                mov     ax, Fn1
                add     ax, Fn2
                pop     Fn2          ;Fn1 becomes the new Fn2 value.
                mov     Fn1, ax      ;Current value becomes new Fn1 value.
                yield                ;Yield the current value.

                inc     wp CurIndex
                jmp     WhlLp



FailFib:        mov     sp, bp       ;Deallocate local vars.
                pop     bp           ;Restore Dynamic Link.
```

```
                add     sp, 2           ;Skip ret adrs.
                ret     2               ;Return through fail address.
Fib             endp


; UpDown-       This function yields the sequence 0, 1, 2, ..., n, n-1,
;               n-2, ..., 1, 0.

i               textequ <[bp-2]>        ;F(n-2) value.

UpDown          proc
                push    bp
                mov      bp, sp
                sub      sp, 2          ;Make room for i.

                mov      wp i, 0        ;Initialize our index variable (i).
UptoN:          mov      ax, i
                cmp      ax, n
                jae      GoDown

                yield

                inc      wp i
                jmp      UpToN

GoDown:         mov      ax, i
                yield
                mov      ax, i
                cmp      ax, 0
                je       UpDownDone
                dec      wp i
                jmp      GoDown

UpDownDone:     mov      sp, bp         ;Deallocate local vars.
                pop      bp             ;Restore Dynamic Link.
                add      sp, 2          ;Skip ret adrs.
                ret      2              ;Return through fail address.
UpDown          endp



; SumToN(n)-    This iterator returns 1, 2, 3, 6, 10, ... sum(n) where
;               sum(n) = 1+2+3+4+...+n  (e.g., n(n+1)/2);

j               textequ <[bp-2]>
k               texteq  <[bp-4]>

SumToN          proc
                push     bp
                mov      bp, sp
                sub      sp, 4          ;Make room for j and k.

                mov      wp j, 0        ;Initialize our index variable (j).
                mov      wp k, 0        ;Initialize our sum (k).
SumLp:          mov      ax, j
                cmp      ax, n
                ja       SumDone

                add      ax, k
                mov      k, ax

                yield

                inc      wp j
                jmp      SumLp

SumDone:        mov     sp, bp          ;Deallocate local vars.
                pop     bp              ;Restore Dynamic Link.
                add     sp, 2           ;Skip ret adrs.
                ret     2               ;Return through fail address.
```

```
SumToN          endp



; MultiIter- This iterator returns a pointer to each of the above iterators.

MultiIter       proc
                push    bp
                mov     bp, sp

                mov     ax, ofs Fib
                yield
                mov     ax, ofs UpDown
                yield
                mov     ax, ofs SumToN
                yield

                pop     bp
                add     sp, 2
                ret
MultiIter       endp


Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; foreach bx in 0..5 do

                mov     bx, 0         ;Loop control variable for outer loop.
WhlBXle5:

; foreach ax in MultiIter do

                push    ofs MultiDone ;Failure address.
                call    MultiIter     ;Get iterator to call.


; foreach i in [ax](bx) do

                push    bx            ;Push "n" (bx) onto the stack.
                push    ofs IterDone ;Failure Address
                call    ax            ;Call the iterator pointed at by the
;                                     ; return value from MultiIter.
;
; write(ax:3);

                mov     cx, 3
                putisize
                ret

; endfor, writeln;

IterDone:       putcr                 ;Writeln;
                ret

; endfor, writeln;

MultiDone:      putcr
                inc     bx
                cmp     bx, 5
                jbe     WhlBXle5

; endfor

Quit:           ExitPgm               ;DOS macro to quit program.
```

```
Main            endp

cseg           ends


sseg            segment  para stack 'stack'
stk             word     1024 dup (0)
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.7.2 The 80x86 Enter and Leave Instructions

The following code (Ex12_2.asm on the companion CD-ROM) uses the 80x86 enter and leave instructions to maintain a display in a block structured program. Assemble this program with the "/Zi" option and load it into CodeView. Set breakpoints on the calls to the Lex1, Lex2, Lex3, and Lex4 procedures. Run the program and when you encounter a breakpoint, use the F8 key to single step into each procedure. Single step over the enter instruction (to the following nop). Note the values of the bp and sp register before and after the execution of the enter instruction.

For your lab report: explain the values in the bp and sp registers after executing each enter instruction. Dump memory from ss:sp to about ss:sp+32 using a memory window or the dw command in the command window. Describe the contents of the stack after the execution of each enter instruction.

After executing through the enter instruction in the Lex4 procedure, set a breakpoint on each of the leave instructions. Run the program at full speed (using the F5 key) until you hit each of these leave instructions. Note the values of the bp and sp registers before and after the execution of each leave instruction. **For your lab report:** include these bp/sp values in your lab report and explain them.

```
; EX12_2.asm
;
; Program to demonstrate the ENTER and LEAVE instructions in Chapter 12.
;
; This program simulates the following Pascal code:
;
; program EnterLeave;
; var i:integer;
;
;    procedure Lex1;
;    var j:integer;
;
;      procedure Lex2;
;      var k:integer;
;
;          procedure Lex3;
;          var m:integer;
;
;                procedure Lex4;
;                var n:integer;
;                begin
;
;                        writeln('Lex4');
;                        for i:= 0 to 3 do
;                            for j:= 0 to 2 do
;                          write('(',i,',',j,') ');
;                        writeln;
;                        for k:= 1 downto 0 do
;                          for m:= 1 downto 0 do
;                            for n := 0 to 1 do
;                          write('(',m,',',k,',',n,') ');
```

```
;                       writeln;
;               end;
;
;          begin {Lex3}
;
;               writeln('Lex3');
;               for i := 0 to 1 do
;                   for j := 0 to 1 do
;                        for k := 0 to 1 do
;                            for m := 0 to 1 do
;                          writeln(i,j,k,m);
;
;               Lex4;
;
;          end; {Lex3}
;
;      begin {Lex2}
;
;          writeln('Lex2');
;          for i := 1 downto 0 do
;               for j := 0 to 1 do
;                   for k := 1 downto 0 do
;                        write(i,j,k,' ');
;          writeln;
;
;          Lex3;
;
;      end; {Lex2}
;
;    begin {Lex1}
;
;      writeln('Lex1');
;      Lex2;
;
;    end; {Lex1}
;
; begin {Main (lex0)}
;
;      writeln('Main Program');
;      Lex1;
;
; end.
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

                .286                            ;Allow ENTER & LEAVE.

; Common equates all the procedures use:

wp              texteq  <word ptr>
disp1           texteq  <word ptr [bp-2]>
disp2           texteq  <word ptr [bp-4]>
disp3           texteq  <word ptr [bp-6]>

; Note: the data segment and the stack segment are one and the same in this
; program.  This is done to allow the use of the [bx] addressing mode when
; referencing local and intermediate variables without having to use a
; stack segment prefix.

sseg            segment  para stack 'stack'

i               word     ?                      ;Main program variable.
stk             word     2046 dup (0)

sseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:sseg, ss:sseg

; Main's activation record looks like this:
;
```

```
;       | return address |<- SP, BP
;       |----------------|

Main            proc
                mov     ax, ss          ;Make SS=DS to simplify addressing
                mov     ds, ax          ; (there will be no need to stick "SS:"
                mov     es, ax          ; in front of addressing modes like
                                        ; "[bx]").

                print
                byte    "Main Program",cr,lf,0
                call    Lex1
Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

; Lex1's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  | Display
;       |----------------|
;       | J Local var    | <- SP  (BP-4)
;       |----------------|

Lex1_J          textequ  <word ptr [bx-4]>

Lex1            proc    near
                enter   2, 1            ;A 2 byte local variable at lex level 1.
                nop                     ;Spacer instruction for single stepping

                print
                byte    "Lex1",cr,lf,0
                call    Lex2
                leave
                ret
Lex1            endp

; Lex2's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  |
;       |----------------|  | Display
;       | Lex2's AR Ptr  |  |
;       |----------------|
;       | K Local var    | <- SP (BP-6)
;       |----------------|
;
;          writeln('Lex2');
;          for i := 1 downto 0 do
;                for j := 0 to 1 do
;                    for k := 1 downto 0 do
;                         write(i,j,k,' ');
;          writeln;
;
;          Lex3;

Lex2_k          textequ  <word ptr [bx-6]>
k               textequ  <word ptr [bp-6]>

Lex2            proc    near
                enter   2, 2            ;A 2-byte local variable at lex level 2.

                nop                     ;Spacer instruction for single stepping

                print
                byte    "Lex2",cr,lf,0

                mov     i, 1
```

```
ForLpI:         mov     bx, disp1       ;"J" is at lex level one.
                mov     Lex1_J, 0
ForLpJ:         mov     k, 1            ;"K" is local.

ForLpK:         mov     ax, i
                puti
                mov     bx, disp1
                mov     ax, Lex1_J
                puti
                mov     ax, k
                puti
                mov     al, ' '
                putc

                dec     k               ;Decrement from 1->0 and quit
                jns     ForLpK          ; if we hit -1.

                mov     bx, disp1
                inc     Lex1_J
                cmp     Lex1_J, 2
                jb      ForLpJ

                dec     i
                jns     ForLpI

                putcr
                call    Lex3

                leave
                ret
Lex2            endp

; Lex3's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  |
;       |----------------|  |
;       | Lex2's AR Ptr  |  | Display
;       |----------------|  |
;       | Lex3's AR Ptr  |  |
;       |----------------|
;       | M Local var    | <- SP (BP-8)
;       |----------------|
;
;               writeln('Lex3');
;               for i := 0 to 1 do
;                   for j := 0 to 1 do
;                       for k := 0 to 1 do
;                           for m := 0 to 1 do
;                             writeln(i,j,k,m);
;
;               Lex4;

Lex3_M          textequ <word ptr [bx-8]>
m               textequ <word ptr [bp-8]>

Lex3            proc    near
                enter   2, 3            ;2-byte variable at lex level 3.

                nop                     ;Spacer instruction for single stepping

                print
                byte    "Lex3",cr,lf,0

                mov     i, 0
ForILp:         mov     bx, disp1
                mov     Lex1_J, 0
ForJlp:         mov     bx, disp2
                mov     Lex2_K, 0
ForKLp:         mov     m, 0
ForMLp:         mov     ax, i
```

```
                        puti
                        mov     bx, disp1
                        mov     ax, Lex1_J
                        puti
                        mov     bx, disp2
                        mov     ax, Lex2_k
                        puti
                        mov     ax, m
                        puti
                        putcr

                        inc     m
                        cmp     m, 2
                        jb      ForMLp

                        mov     bx, disp2
                        inc     Lex2_K
                        cmp     Lex2_K, 2
                        jb      ForKLp

                        mov     bx, disp1
                        inc     Lex1_J
                        cmp     Lex1_J, 2
                        jb      ForJLp

                        inc     i
                        cmp     i, 2
                        jb      ForILp

                        call    Lex4

                        leave
                        ret
Lex3                    endp

; Lex4's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  |
;       |----------------|  |
;       | Lex2's AR Ptr  |  |
;       |----------------|  | Display
;       | Lex3's AR Ptr  |  |
;       |----------------|  |
;       | Lex4's AR Ptr  |  |
;       |----------------|
;       | N Local var    | <- SP (BP-10)
;       |----------------|
;
;
;       writeln('Lex4');
;       for i:= 0 to 3 do
;         for j:= 0 to 2 do
;                 write('(',i,',',j,') ');
;       writeln;
;       for k:= 1 downto 0 do
;         for m:= 1 downto 0 do
;           for n := 0 to 1 do
;                 write('(',m,',',k,',',n,') ');
;       writeln;

n                       textequ  <word ptr [bp-10]>

Lex4                    proc     near
                        enter    2, 4         ;2-byte local variable at lex level 4.

                        nop                   ;Spacer instruction for single stepping

                        print
                        byte     "Lex4",cr,lf,0
```

```
                mov     i, 0
ForILp:         mov     bx, disp1
                mov     Lex1_J, 0
ForJLp:         mov     al, '('
                putc
                mov     ax, i
                puti
                mov     al, ','
                putc
                mov     ax, Lex1_J   ;Note that BX still contains disp1.
                puti
                print
                byte    ") ",0

                inc     Lex1_J       ;BX still contains disp1.
                cmp     Lex1_J, 3
                jb      ForJLp

                inc     i
                cmp     i, 4
                jb      ForILp

                putcr

                mov     bx, disp2
                mov     Lex2_K, 1
ForKLp:         mov     bx, disp3
                mov     Lex3_M, 1
ForMLp:         mov     n, 0
ForNLp:         mov     al, '('
                putc

                mov     bx, disp3
                mov     ax, Lex3_M
                puti
                mov     al, ','
                putc
                mov     bx, disp2
                mov     ax, Lex2_K
                puti
                mov     al, ','
                putc
                mov     ax, n
                puti
                print
                byte    ") ",0

                inc     n
                cmp     n, 2
                jb      ForNLp

                mov     bx, disp3
                dec     Lex3_M
                jns     ForMLp

                mov     bx, disp2
                dec     Lex2_K
                jns     ForKLp

                leave
                ret
Lex4            endp
cseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.7.3 Parameter Passing Exercises

The following exercise demonstrates some simple parameter passing. This program passes arrays by reference, word variables by value and by reference, and some functions and procedure by reference. The program itself sorts two arrays using a generic sorting algorithm. The sorting algorithm is generic because the main program passes it a comparison function and a procedure to swap two elements if one is greater than the other.

```
; Ex12_3.asm
;
; This program demonstrates different parameter passing methods.
; It corresponds to the following (pseudo) Pascal code:
;
;
; program main;
; var  i:integer;
;      a:array[0..255] of integer;
;      b:array[0..255] of unsigned;
;
; function LTint(int1, int2:integer):boolean;
; begin
;      LTint := int1 < int2;
; end;
;
; procedure SwapInt(var int1, int2:integer);
; var temp:integer;
; begin
;      temp := int1;
;      int1 := int2;
;      int2 := temp;
; end;
;
; function LTunsigned(uns1, uns2:unsigned):boolean;
; begin
;      LTunsigned := uns1 < uns2;
; end;
;
; procedure SwapUnsigned(uns1, uns2:unsigned);
; var temp:unsigned;
; begin
;      temp := uns1;
;      uns1 := uns2;
;      uns2 := temp;
; end;
;
; (* The following is a simple Bubble sort that will sort arrays containing *)
; (* arbitrary data types.                                                   *)
;
; procedure sort(data:array; elements:integer; function LT:boolean; procedure
swap);
; var i,j:integer;
; begin
;
;      for i := 0 to elements-1 do
;          for j := i+1 to elements do
;                if (LT(data[j], data[i])) then swap(data[i], data[j]);
; end;
;
;
; begin
;
;      for i := 0 to 255 do A[i] := 128-i;
;      for i := 0 to 255 do B[i] := 255-i;
;      sort(A, 256, LTint, SwapInt);
;      sort(B, 256, LTunsigned, SwapUnsigned);
;
;      for i := 0 to 255 do
;      begin
```

```
;               if (i mod 8) = 0 then writeln;
;               write(A[i]:5);
;       end;
;
;       for i := 0 to 255 do
;       begin
;               if (i mod 8) = 0 then writeln;
;               write(B[i]:5);
;       end;
;
; end;

                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

                .386
                option      segment:use16

wp              texequ      <word ptr>

dseg            segment     para public 'data'
A               word        256 dup (?)
B               word        256 dup (?)
dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg, ss:sseg

; function LTint(int1, int2:integer):boolean;
; begin
;       LTint := int1 < int2;
; end;
;
; LTint's activation record looks like this:
;
;       |----------------|
;       |      int1      |
;       |----------------|
;       |      int2      |
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|

int1            texequ      <word ptr [bp+6]>
int2            texequ      <word ptr [bp+4]>

LTint           proc        near
                push        bp
                mov         bp, sp

                mov         ax, int1        ;Compare the two parameters
                cmp         ax, int2        ; and return true if int1<int2.
                setl        al              ;Signed comparison here.
                mov         ah, 0           ;Be sure to clear H.O. byte.

                pop         bp
                ret         4
LTint           endp

; Swap's activation record looks like this:
;
;       |----------------|
;       |     Address    |
;       |---    of    ---|
;       |      int1      |
;       |----------------|
;       |     Address    |
;       |---    of    ---|
;       |      int2      |
```

```
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|
;
; The temporary variable is kept in a register.
;
; Note that swapping integers or unsigned integers can be done
; with the same code since the operations are identical for
; either type.
;
; procedure SwapInt(var int1, int2:integer);
; var temp:integer;
; begin
;       temp := int1;
;       int1 := int2;
;       int2 := temp;
; end;
;
; procedure SwapUnsigned(uns1, uns2:unsigned);
; var temp:unsigned;
; begin
;       temp := uns1;
;       uns1 := uns2;
;       uns2 := temp;
; end;
;
int1            textequ  <dword ptr [bp+8]>
int2            textequ  <dword ptr [bp+4]>

SwapInt         proc     near
                push     bp
                mov      bp, sp
                push     es
                push     bx

                les      bx, int1     ;Get address of int1 variable.
                mov      ax, es:[bx]  ;Get int1's value.
                les      bx, int2     ;Get address of int2 variable.
                xchg     ax, es:[bx]  ;Swap int1's value with int2's

                les      bx, int1     ;Get the address of int1 and
                mov      es:[bx], ax  ; store int2's value there.

                pop      bx
                pop      es
                pop      bp
                ret      8
SwapInt         endp

; LTunsigned's activation record looks like this:
;
;       |----------------|
;       |      uns1      |
;       |----------------|
;       |      uns2      |
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|
;
; function LTunsigned(uns1, uns2:unsigned):boolean;
; begin
;       LTunsigned := uns1 < uns2;
; end;

uns1            textequ  <word ptr [bp+6]>
uns2            textequ  <word ptr [bp+4]>

LTunsigned      proc     near
```

```
                push    bp
                mov     bp, sp

                mov     ax, uns1        ;Compare uns1 with uns2 and
                cmp     ax, uns2        ; return true if uns1<uns2.
                setb    al              ;Unsigned comparison.
                mov     ah, 0           ;Return 16-bit boolean.

                pop     bp
                ret     4
LTunsigned      endp

; Sort's activation record looks like this:
;
;       |----------------|
;       |     Data's     |
;       |---           ---|
;       |    Address     |
;       |----------------|
;       |   Elements     |
;       |----------------|
;       |      LT's      |
;       |---           ---|
;       |    Address     |
;       |----------------|
;       |     Swap's     |
;       |---           ---|
;       |    Address     |
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|
;
; procedure sort(data:array; elements:integer; function LT:boolean; procedure
swap);
; var i,j:integer;
; begin
;
;       for i := 0 to elements-1 do
;           for j := i+1 to elements do
;                 if (LT(data[j], data[i])) then swap(data[i], data[j]);
; end;

data            textequ  <dword ptr [bp+10]>
elements        textequ  <word ptr [bp+8]>
funcLT          textequ  <word ptr [bp+6]>
procSwap        textequ  <word ptr [bp+4]>

i               textequ  <word ptr [bp-2]>
j               textequ  <word ptr [bp-4]>

sort            proc     near
                push     bp
                mov      bp, sp
                sub      sp, 4
                push     es
                push     bx

                mov      i, 0
ForILp:         mov      ax, i
                inc      i
                cmp      ax, Elements
                jae      IDone

                mov      j, ax
ForJLp:         mov      ax, j
                cmp      ax, Elements
                ja       JDone

                les      bx, data               ;Push the value of
                mov      si, j                  ; data[j] onto the
                add      si, si                 ; stack.
```

```
                        push    es:[bx+si]

                        les     bx, data                ;Push the value of
                        mov     si, i                   ; data[i] onto the
                        add     si, si                  ; stack.
                        push    es:[bx+si]

                        call    FuncLT                  ;See if data[i] < data[j]
                        cmp     ax, 0                   ;Test boolean result.
                        je      NextJ

                        push    wp data+2               ;Pass data[i] by reference.
                        mov     ax, i
                        add     ax, ax
                        add     ax, wp data
                        push    ax

                        push    wp data+2               ;Pass data[j] by reference.
                        mov     ax, j
                        add     ax, ax
                        add     ax, wp data
                        push    ax

                        call    ProcSwap

NextJ:                  inc     j
                        jmp     ForJLp

JDone:                  inc     i
                        jmp     ForILp

IDone:                  pop     bx
                        pop     es
                        mov     sp, bp
                        pop     bp
                        ret     10
sort                    endp

; Main's activation record looks like this:
;
;       | return address |<- SP, BP
;       |----------------|
;
; begin
;
;       for i := 0 to 255 do A[i] := 128-i;
;       for i := 0 to 255 do B[i] := 33000-i;
;       sort(A, 256, LTint, SwapInt);
;       sort(B, 256, LTunsigned, SwapUnsigned);
;
;       for i := 0 to 255 do
;       begin
;                 if (i mod 8) = 0 then writeln;
;                 write(A[i]:5);
;       end;
;
;       for i := 0 to 255 do
;       begin
;                 if (i mod 8) = 0 then writeln;
;                 write(B[i]:5);
;       end;
;
; end;

Main                    proc
                        mov     ax, dseg        ;Initialize the segment registers.
                        mov     ds, ax
                        mov     es, ax

; Note that the following code merges the two initialization for loops
; into a single loop.

                        mov     ax, 128
                        mov     bx, 0
```

```
                mov     cx, 33000
ForILp:         mov     A[bx], ax
                mov     B[bx], cx
                add     bx, 2
                dec     ax
                dec     cx
                cmp     bx, 256*2
                jb      ForILp

                push    ds                      ;Seg address of A
                push    offset A                ;Offset of A
                push    256                     ;# of elements in A
                push    offset LTint            ;Address of compare routine
                push    offset SwapInt          ;Address of swap routine
                call    Sort

                push    ds                      ;Seg address of B
                push    offset B                ;Offset of B
                push    256                     ;# of elements in A
                push    offset LTunsigned       ;Address of compare routine
                push    offset SwapInt          ;Address of swap routine
                call    Sort

; Print the values in A.

                mov     bx, 0
ForILp2:        test    bx, 0Fh                 ;See if (I mod 8) = 0
                jnz     NotMod                  ; note: BX mod 16 = I mod 8.
                putcr
NotMod:         mov     ax, A[bx]
                mov     cx, 5
                putisize
                add     bx, 2
                cmp     bx, 256*2
                jb      ForILp2

; Print the values in B.
                mov     bx, 0
ForILp3:        test    bx, 0Fh                 ;See if (I mod 8) = 0
                jnz     NotMod2                 ; note: BX mod 16 = I mod 8.
                putcr
NotMod2:        mov     ax, B[bx]
                mov     cx, 5
                putusize
                add     bx, 2
                cmp     bx, 256*2
                jb      ForILp3

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             word     256 dup (0)
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.8 Programming Projects

1) Write at iterator to which you pass an array of characters by reference. The iterator should return an index into the array that points at a whitespace character (any ASCII code less than or equal to a space) it finds. On each call, the iterator should return the index of the next whitespace character. The iterator should fail if it encounters a byte containing the value zero. Use local variables for any values the iterator needs.

2) Write a recursive routine that does the following:

```
function recursive(i:integer):integer;
var j,k:integer;
begin
          j := i;
          k := i*i;
          if (i >= 0) then writeln('AR Address =', Recursive(i-1));
          writeln(i,' ',j,' ',k);
          recursive := Value in BP Register;
end;
```

From your main program, call this procedure and pass it the value 10 on the stack. Verify that you get correct results back. Explain the results.

3) Write a program that contains a procedure to which you pass four parameters on the stack. These should be passed by value, reference, value-result, and result, respectively (for the value-result parameter, pass the address of the object on the stack). Inside that procedure, you should call three other procedures that also take four parameters (each). However, the first parameter should use pass by value for all four parameters; the second procedure should use pass by reference for all four parameters; and the third should use pass by value-result for all four parameters. Pass the four parameters in the enclosing procedure as parameters to each of these three child procedures. Inside the child procedures, print the parameter's values and change their results. Immediately upon return from each of these child procedures, print the parameters' values. Write a main program that passes four local (to the main program) variables you've initialized with different values to the first procedure above. Run the program and verify that it is operating correctly and that it is passing the parameters to each of these procedures in a reasonable fashion.

4) Write a program that implements the following Pascal program in assembly language. Assume that all program variables (including globals in the main program) are allocated in activation records on the stack.

```
program nest3;
var    i:integer;

       procedure A(k:integer);

                 procedure B(procedure c);
                 var m:integer;
                 begin

                          for m:= 0 to 4 do c(m);

                 end; {B}

                 procedure D(n:integer);
                 begin

                          for i:= 0 to n-1 do writeln(i);

                 end; {D}

                 procedure E;
                 begin

                          writeln('A stuff:');
                          B(A);
                          writeln('D stuff:');
                          B(D);

                 end; {E}

       begin {A}

                 B(D);
                 writeln;
                 if k < 2 then E;
```

```
        end; {A}

begin {nest3}

        A(0);

end; {nest3}
```

5) The program in Section 12.7.2 (Ex12_2.asm on the companion CD-ROM) uses the 80286 enter and leave instructions to maintain the display in each activation record. As pointed out in Section 12.1.6, these instructions are quite slow, especially on 80486 and later processors. Rewrite this code by replacing the enter and leave instructions with the straight-line code that does the same job. In CodeView, single step through the program as per the second laboratory exercise (Section 12.7.2) to verify that your stack frames are identical to those the enter and leave instructions produce.

6) The generic Bubble Sort program in Section 12.7.3 only works with data objects that are two bytes wide. This is because the Sort procedure passes the values of Data[I] and Data[J] on the stack to the comparison routines (LTint and LTunsigned) and because the sort routine multiplies the i and j indexes by two when indexing into the data array. This is a severe shortcoming to this generic sort routine. Rewrite the program to make it truly generic. Do this by writing a "CompareAndSwap" routine that will replace the LT and Swap calls. To CompareAndSwap you should pass the array (by reference) and the two array indexes (i and j) to compare and possibly swap. Write two versions of the CompareAndSwap routine, one for unsigned integers and one for signed integers. Run this program and verify that your implementation works properly.

12.9 Summary

Block structured languages, like Pascal, provide access to non-local variables at different lex levels. Accessing non-local variables is a complex task requiring special data structures such as a static link chain or a display. The display is probably the most efficient way to access non-local variables. The 80286 and later processors provide special instructions, enter and leave for maintaining a display list, but these instructions are too slow for most common uses. For additional details, see

- "Lexical Nesting, Static Links, and Displays" on page 639
- "Scope" on page 640
- "Static Links" on page 642
- "Accessing Non-Local Variables Using Static Links" on page 647
- "The Display" on page 648
- "The 80286 ENTER and LEAVE Instructions" on page 650
- "Passing Variables at Different Lex Levels as Parameters." on page 652
- "Passing Parameters as Parameters to Another Procedure" on page 655
- "Passing Procedures as Parameters" on page 659

Iterators are a cross between a function and a looping construct. They are a very powerful programming construct available in many very high level languages. Efficient implementation of iterators involves careful manipulation of the stack at run time. To see how to implement iterators, read the following sections:

- "Iterators" on page 663
- "Implementing Iterators Using In-Line Expansion" on page 664
- "Implementing Iterators with Resume Frames" on page 666
- "An Example of an Iterator" on page 669
- "Another Iterator Example" on page 673

12.10 Questions

1) What is an iterator?

2) What is a resume frame?

3) How do the iterators in this chapter implement the success and failure results?

4) What does the stack look like when executing the body of a loop controlled by an iterator?

5 What is a static link?

6) What is a display?

7) Describe how to access a non-local variable when using static links.

8) Describe how to access a non-local variable when using a display.

9) How would you access a non-local variable when using the display built by the 80286 ENTER instruction?

10) Draw a picture of the activation record for a procedure at lex level 4 that uses the ENTER instruction to build the display.

11) Explain why the static links work better than a display when passing procedures and functions as parameters.

12) Suppose you want to pass an intermediate variable by value-result using the technique where you push the value before calling the procedure and then pop the value (storing it back into the intermediate variable) upon return from the procedure. Provide two examples, one using static links and one using a display, that implement pass by value-result in this fashion.

13) Convert the following (pseudo) Pascal code into 80x86 assembly language. Assume Pascal supports pass by name and pass by lazy evaluation parameters as suggested by the following code.

```
program main;
var k:integer;

procedure one(LazyEval i:integer);
begin
       writeln(i);
end;

procedure two(name j:integer);
begin
       one(j);
end;

begin {main}
       k := 2;
       two(k);
end;
```

A typical PC system consists of many component besides the 80x86 CPU and memory. MS-DOS and the PC's BIOS provide a software connection between your application program and the underlying hardware. Although it is sometimes necessary to program the hardware directly yourself, more often than not it's best to let the system software (MS-DOS and the BIOS) handle this for you. Furthermore, it's much easier for you to simply call a routine built into your system than to write the routine yourself.

You can access the IBM PC system hardware at one of three general levels from assembly language. You can program the hardware directly, you can use ROM BIOS routines to access the hardware for you, or you can make MS-DOS calls to access the hardware. Each level of system access has its own set of advantages and disadvantages.

Programming the hardware directly offers two advantages over the other schemes: control and efficiency. If you're controlling the hardware modes, you can get that last drop of performance out of the system by taking advantage of special hardware tricks or other details which a general purpose routine cannot. For some programs, like screen editors (which must have high speed access to the video display), accessing the hardware directly is the only way to achieve reasonable performance levels.

On the other hand, programming the hardware directly has its drawbacks as well. The screen editor which directly accesses video memory may not work if a new type of video display card appears for the IBM PC. Multiple display drivers may be necessary for such a program, increasing the amount of work to create and maintain the program. Furthermore, had you written several programs which access the screen memory directly and IBM produced a new, incompatible, display adapter, you'd have to rewrite all your programs to work with the new display card.

Your work load would be reduced tremendously if IBM supplied, in a fixed, known, location, some routines which did all the screen I/O operations for you. Your programs would all call these routines. When a manufacturer introduces a new display adapter, it supplies a new set of video display routines with the adapter card. These new routines would patch into the old ones (replacing or augmenting them) so that calls to the old routines would now call the new routines. If the program interface is the same between the two set of routines, your programs will still work with the new routines.

IBM has implemented such a mechanism in the PC system firmware. Up at the high end of the one megabyte memory space in the PC are some addresses dedicated to ROM data storage. These ROM memory chips contain special software called the PC Basic Input Output System, or BIOS. The BIOS routines provide a hardware-independent interface to various devices in the IBM PC system. For example, one of the BIOS services is a video display driver. By making various calls to the BIOS video routines, your software will be able to write characters to the screen regardless of the actual display board installed.

At one level up is MS-DOS. While the BIOS allows you to manipulate devices in a very low level fashion, MS-DOS provides a high-level interface to many devices. For example, one of the BIOS routines allows you to access the floppy disk drive. With this BIOS routine you may read or write blocks on the diskette. Unfortunately, the BIOS doesn't know about things like files and directories. It only knows about blocks. If you want to access a file on the disk drive using a BIOS call, you'll have to know exactly where that file appears on the diskette surface. On the other hand, calls to MS-DOS allow you to deal with filenames rather than file disk addresses. MS-DOS keeps track of where files are on the disk surface and makes calls to the ROM BIOS to read the appropriate blocks for you. This high-level interface greatly reduces the amount of effort your software need expend in order to access data on the disk drive.

The purpose of this chapter is to provide a brief introduction to the various BIOS and DOS services available to you. This chapter does not attempt to begin to describe all of the routines or the options available to each routine. There are several other texts the size of this one which attempt to discuss *just* the BIOS or *just* MS-DOS. Furthermore, any attempt

to provide complete coverage of MS-DOS or the BIOS in a single text is doomed to failure from the start– both are a moving target with specifications changing with each new version. So rather than try to explain everything, this chapter will simply attempt to present the flavor. Check in the bibliography for texts dealing directly with BIOS or MS -DOS.

13.0 Chapter Overview

This chapter presents material that is specific to the PC. This information on the PC's BIOS and MS-DOS is not necessary if you want to learn about assembly language programming; however, this is important information for anyone wanting to write assembly language programs that run under MS-DOS on a PC compatible machine. As a result, most of the information in this chapter is optional for those wanting to learn generic 80x86 assembly language programming. On the other hand, this information is handy for those who want to write applications in assembly language on a PC.

The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- The IBM PC BIOS
- ❑ Print screen.
- Video services.
- ❑ Equipment installed.
- ❑ Memory available.
- ❑ Low level disk services
- Serial I/O.
- ❑ Miscellaneous services.
- Keyboard services.
- Printer services.
- ❑ Run BASIC.
- ❑ Reboot computer.
- ❑ Real time clock.
- MS-DOS calling sequence.
- MS-DOS character functions
- ❑ MS-DOS drive commands.
- ❑ MS-DOS date and time functions.
- ❑ MS-DOS memory management functions.
- ❑ MS-DOS process control functions.
- MS_DOS "new" filing calls.
- Open file.
- Create file.
- Close file.
- Read from a file.
- Write to a file.
- ❑ Seek.
- ❑ Set disk transfer address.
- ❑ Find first file.
- ❑ Find next file.
- Delete file.
- Rename file.
- ❑ Change/get file attributes.
- ❑ Get/set file date and time.
- ❑ Other DOS calls
- File I/O examples.
- Blocked file I/O.
- ❑ The program segment prefix.
- ❑ Accessing command line parameters.
- ❑ ARGC and ARGV.
- UCR Standard Library file I/O routines.

- FOPEN.
- FCREATE.
- FCLOSE.
- FFLUSH.
- FGETC.
- FREAD.
- FPUTC
- FWRITE.

❑ Redirection I/O through the STDLIB file I/O routines.

13.1 The IBM PC BIOS

Rather than place the BIOS routines at fixed memory locations in ROM, IBM used a much more flexible approach in the BIOS design. To call a BIOS routine, you use one of the 80x86's int software interrupt instructions. The int instruction uses the following syntax:

int *value*

Value is some number in the range 0..255. Execution of the int instruction will cause the 80x86 to transfer control to one of 256 different interrupt handlers. The interrupt vector table, starting at physical memory location 0:0, holds the addresses of these interrupt handlers. Each address is a full segmented address, requiring four bytes, so there are 400h bytes in the interrupt vector table -- one segmented address for each of the 256 possible software interrupts. For example, int 0 transfers control to the routine whose address is at location 0:0, int 1 transfers control to the routine whose address is at 0:4, int 2 via 0:8, int 3 via 0:C, and int 4 via 0:10.

When the PC resets, one of the first operations it does is initialize several of these interrupt vectors so they point at BIOS service routines. Later, when you execute an appropriate int instruction, control transfers to the appropriate BIOS code.

If all you're doing is calling BIOS routines (as opposed to writing them), you can view the int instruction as nothing more than a special call instruction.

13.2 An Introduction to the BIOS' Services

The IBM PC BIOS uses software interrupts 5 and 10h..1Ah to accomplish various operations. Therefore, the int 5, and int 10h.. int 1ah instructions provide the interface to BIOS. The following table summarizes the BIOS services:

INT	Function
5	Print Screen operation.
10h	Video display services.
11h	Equipment determination.
12h	Memory size determination.
13h	Diskette and hard disk services.
14h	Serial I/O services.
15h	Miscellaneous services.
16h	Keyboard services.
17h	Printer services.
18h	BASIC.
19h	Reboot.
1Ah	Real time clock services.

Most of these routines require various parameters in the 80x86's registers. Some require additional parameters in certain memory locations. The following sections describe the exact operation of many of the BIOS routine.

13.2.1 INT 5- Print Screen

Instruction: int 5h
BIOS Operation: Print the current text screen.
Parameters: None

If you execute the int 5h instruction, the PC will send a copy of the screen image to the printer exactly as though you'd pressed the PrtSc key on the keyboard. In fact, the BIOS issues an int 5 instruction when you press the PrtSc, so the two operations are absolutely identical (other than one is under software control rather than manual control). Note that the 80286 and later also uses int 5 for the BOUNDS trap.

13.2.2 INT 10h - Video Services

Instruction: int 10h
BIOS Operation: Video I/O Services
Parameters: Several, passed in ax, bx, cx, dx, and es:bp registers.

The int 10h instruction does several video display related functions. You can use it to initialize the video display, set the cursor size and position, read the cursor position, manipulate a light pen, read or write the current display page, scroll the data in the screen up or down, read and write characters, read and write pixels in a graphics display mode, and write strings to the display. You select the particular function to execute by passing a value in the ah register.

The video services represent one of the largest set of BIOS calls available. There are many different video display cards manufactured for PCs, each with minor variations and often each having its own set of unique BIOS functions. The BIOS reference in the appendices lists some of the more common functions available, but as pointed out earlier, this list is quite incomplete and out of date given the rapid change in technology.

Probably the most commonly used video service call is the character output routine:

Name: Write char to screen in TTY mode
Parameters ah = 0Eh, al = ASCII code (In graphics mode, bl = Page number)

This routine writes a single character to the display. MS-DOS calls this routine to display characters on the screen. The UCR Standard Library also provides a call which lets you write characters directly to the display using BIOS calls.

Most BIOS video display routines are poorly written. There is not much else that can be said about them. They are extremely slow and don't provide much in the way of functionality. For this reason, most programmers (who need a high-performance video display driver) end up writing their own display code. This provides speed at the expense of portability. Unfortunately, there is rarely any other choice. If you need functionality rather than speed, you should consider using the ANSI.SYS screen driver provided with MS-DOS. This display driver provides all kinds of useful services such as clear to end of line, clear to end of screen, etc. For more information, consult your DOS manual.

Table 49: BIOS Video Functions (Partial List)

AH	Input Parameters	Output Parameters	Description
0	al=mode		Sets the video display mode.
1	ch- Starting line. cl- ending line		Sets the shape of the cursor. Line values are in the range 0..15. You can make the cursor disappear by loading ch with 20h.

Table 49: BIOS Video Functions (Partial List)

AH	Input Parameters	Output Parameters	Description
2	bh- page dh- y coordinate dl- x coordinate		Position cursor to location (x,y) on the screen. Generally you would specify page zero. BIOS maintains a separate cursor for each page.
3	bh- page	ch- starting line cl- ending line dl- x coordinate dh- y coordinate	Get cursor position and shape.
4			Obsolete (Get Light Pen Position).
5	al- display page		Set display page. Switches the text display page to the specified page number. Page zero is the standard text page. Most color adapters support up to eight text pages (0..7).
6	al- Number of lines to scroll. bh- Screen attribute for cleared area. cl- x coordinate UL ch- y coordinate UL dl- x coordinate LR dh- y coordinate LR		Clear or scroll up. If al contains zero, this function clears the rectangular portion of the screen specified by cl/ch (the upper left hand corner) and dl/dh (the lower right hand corner). If al contains any other value, this service will scroll that rectangular window up the number of lines specified in al.
7	al- Number of lines to scroll. bh- Screen attribute for cleared area. cl- x coordinate UL ch- y coordinate UL dl- x coordinate LR dh- y coordinate LR		Clear or scroll down. If al contains zero, this function clears the rectangular portion of the screen specified by cl/ch (the upper left hand corner) and dl/dh (the lower right hand corner). If al contains any other value, this service will scroll that rectangular window down the number of lines specified in al.
8	bh- display page	al- char read ah- char attribute	Read character's ASCII code and attribute byte from current screen position.
9	al- character bh- page bl- attribute cx- # of times to replicate character		This call writes cx copies of the character and attribute in al/bl starting at the current cursor position on the screen. It does not change the cursor's position.
0Ah	al- character bh- page		Writes character in al to the current screen position using the existing attribute. Does not change cursor position.
0Bh	bh- 0 bl- color		Sets the border color for the text display.
0Eh	al- character bh- page		Write a character to the screen. Uses existing attribute and repositions cursor after write.
0Fh		ah- # columns al- display mode bh- page	Get video mode

Note that there are many other BIOS 10h subfunctions. Mostly, these other functions deal with graphics modes (the BIOS is too slow for manipulating graphics, so you shouldn't use those calls) and extended features for certain video display cards. For more information on these calls, pick up a text on the PC's BIOS.

13.2.3 INT 11h - Equipment Installed

Instruction: int 11h
BIOS Operation: Return an equipment list
Parameters: On entry: None, on exit: AX contains equipment list

On return from int 11h, the AX register contains a bit-encoded equipment list with the following values:

Bit 0	Floppy disk drive installed
Bit 1	Math coprocessor installed
Bits 2,3	System board RAM installed (obsolete)
Bits 4,5	Initial video mode 00- none 01- 40x25 color 10- 80x25 color 11- 80x25 b/w
Bits 6,7	Number of disk drives
Bit 8	DMA present
Bits 9,10,11	Number of RS-232 serial cards installed
Bit 12	Game I/O card installed
Bit 13	Serial printer attached
Bits 14,15	Number of printers attached.

Note that this BIOS service was designed around the original IBM PC with its very limited hardware expansion capabilities. The bits returned by this call are almost meaningless today.

13.2.4 INT 12h - Memory Available

Instruction: int 12h
BIOS Operation: Determine memory size
Parameters: Memory size returned in AX

Back in the days when IBM PCs came with up to 64K memory installed on the motherboard, this call had some meaning. However, PCs today can handle up to 64 *megabytes* or more. Obviously this BIOS call is a little out of date. Some PCs use this call for different purposes, but you cannot rely on such calls working on any machine.

13.2.5 INT 13h - Low Level Disk Services

Instruction: int 13h
BIOS Operation: Diskette Services
Parameters: ax, es:bx, cx, dx (see below)

The int 13h function provides several different low-level disk services to PC programs: Reset the diskette system, get the diskette status, read diskette sectors, write diskette sectors, verify diskette sectors, and format a diskette track and many more. This is another example of a BIOS routine which has changed over the years. When this routine was first developed, a 10 megabyte hard disk was considered large. Today, a typical high performance game requires 20 to 30 megabytes of storage.

Table 50: Some Common Disk Subsystem BIOS Calls

AH	Input Parameters	Output Parameters	Description
0	dl- drive (0..7fh is floppy, 80h..ffh is hard)	ah- status (0 and carry clear if no error, error code if error).	Resets the specified disk drive. Resetting a hard disk also resets the floppy drives.
1	dl- drive (as above)	ah- 0 al- status of previous disk operation.	This call returns the following status values in al: 0- no error 1- invalid command 2- address mark not found 3- disk write protected 4- couldn't find sector 5- reset error 6- removed media 7- bad parameter table 8- DMA overrun 9- DMA operation crossed 64K boundary 10- illegal sector flag 11- illegal track flag 12- illegal media 13- invalid # of sectors 14- control data address mark encountered 15- DMA error 16- CRC data error 17- ECC corrected data error 32- disk controller failed 64- seek error 128- timeout error 170- drive not ready 187- undefined error 204- write error 224- status error 255- sense failure
2	al- # of sectors to read es:bx- buffer address cl- bits 0..5: sector # cl- bits 6/7- track bits 8 & 9 ch- track bits 0..7. dl- drive # (as above) dh- bits 0..5: head # dh- bits 6&7: track bits 10 & 11.	ah- return status al- burst error length carry- 0:success, 1:error	Reads the specified number of 512 byte sectors from the disk. Data read must be 64 Kbytes or less.
3	same as (2) above	same as (2) above	Writes the specified number of 512 byte sectors to the disk. Data written must not exceed 64 Kbytes in length.
4	Same as (2) above except there is no need for a buffer.	same as (2) above	Verifies the data in the specified number of 512 byte sectors on the disk.
0Ch	Same as (4) above except there is no need for a sector #	Same as (4) above	Sends the disk head to the specified track on the disk.

Table 50: Some Common Disk Subsystem BIOS Calls

AH	Input Parameters	Output Parameters	Description
0Dh	dl- drive # (80h or 81h)	ah- return status carry-0:no error 1:error	Reset the hard disk controller

Note: see appropriate BIOS documentation for additional information about disk subsystem BIOS support.

13.2.6 INT 14h - Serial I/O

Instruction: int 14h
BIOS Operation: Access the serial communications port
Parameters: ax, dx

The IBM BIOS supports up to four different serial communications ports (the hardware supports up to eight). In general, most PCs have one or two serial ports (COM1: and COM2:) installed. Int 14h supports four subfunctions- initialize, transmit a character, receive a character, and status. For all four services, the serial port number (a value in the range 0..3) is in the dx register (0=COM1:, 1=COM2:, etc.). Int 14h expects and returns other data in the al or ax register.

13.2.6.1 AH=0: Serial Port Initialization

Subfunction zero initializes a serial port. This call lets you set the baud rate, select parity modes, select the number of stop bits, and the number of bits transmitted over the serial line. These parameters are all specified by the value in the al register using the following bit encodings:

Bits — Function

5..7 — Select baud rate
- 000- 110 baud
- 001- 150
- 010- 300
- 011- 600
- 100- 1200
- 101- 2400
- 110- 4800
- 111- 9600

3..4 — Select parity
- 00- No parity
- 01- Odd parity
- 10- No parity
- 11- Even parity

2 — Stop bits
- 0-One stop bit
- 1-Two stop bits

0..1 — Character Size
- 10- 7 bits
- 11- 8 bits

Although the standard PC serial port hardware supports 19,200 baud, some BIOSes may not support this speed.

Example: Initialize COM1: to 2400 baud, no parity, eight bit data, and two stop bits-

```
mov     ah, 0                   ;Initialize opcode
mov     al, 10100111b           ;Parameter data.
mov     dx, 0                   ;COM1: port.
int     14h
```

After the call to the initialization code, the serial port status is returned in ax (see Serial Port Status, ah=3, below).

13.2.6.2 AH=1: Transmit a Character to the Serial Port

This function transmits the character in the al register through the serial port specified in the dx register. On return, if ah contains zero, then the character was transmitted properly. If bit 7 of ah contains one, upon return, then some sort of error occurred. The remaining seven bits contain all the error statuses returned by the GetStatus call except time out error (which is returned in bit seven). If an error is reported, you should use subfunction three to get the actual error values from the serial port hardware.

Example: Transmit a character through the COM1: port

```
mov     dx, 0       ;Select COM1:
mov     al, 'a'     ;Character to transmit
mov     ah, 1       ;Transmit opcode
int     14h
test    ah, 80h     ;Check for error
jnz     SerialError
```

This function will wait until the serial port finishes transmitting the last character (if any) and then it will store the character into the transmit register.

13.2.6.3 AH=2: Receive a Character from the Serial Port

Subfunction two is used to read a character from the serial port. On entry, dx contains the serial port number. On exit, al contains the character read from the serial port and bit seven of ah contains the error status. When this routine is called, it does not return to the caller until a character is received at the serial port.

Example: Reading a character from the COM1: port

```
mov     dx, 0       ;Select COM1:
mov     ah, 2       ;Receive opcode
int     14h
test    ah, 80h     ;Check for error
jnz     SerialError

<Received character is now in AL>
```

13.2.6.4 AH=3: Serial Port Status

This call returns status information about the serial port including whether or not an error has occurred, if a character has been received in the receive buffer, if the transmit buffer is empty, and other pieces of useful information. On entry into this routine, the dx register contains the serial port number. On exit, the ax register contains the following values:

AX:	Bit Meaning
15	Time out error
14	Transmitter shift register empty
13	Transmitter holding register empty
12	Break detection error
11	Framing error
10	Parity error
9	Overrun error
8	Data available
7	Receive line signal detect
6	Ring indicator
5	Data set ready (DSR)
4	Clear to send (CTS)
3	Delta receive line signal detect
2	Trailing edge ring detector
1	Delta data set ready
0	Delta clear to send

There are a couple of useful bits, not pertaining to errors, returned in this status information. If the data available bit is set (bit #8), then the serial port has received data and you should read it from the serial port. The Transmitter holding register empty bit (bit #13) tells you if the transmit operation will be delayed while waiting for the current character to be transmitted or if the next character will be immediately transmitted. By testing these two bits, you can perform other operations while waiting for the transmit register to become available or for the receive register to contain a character.

If you're interested in serial communications, you should obtain a copy of Joe Campbell's C Programmer's Guide to Serial Communications. Although written specifically for C programmers, this book contains a lot of information useful to programmers working in any programming language. See the bibliography for more details.

13.2.7 INT 15h - Miscellaneous Services

Originally, int 15h provided cassette tape read and write services[1]. Almost immediately, everyone realized that cassettes were history, so IBM began using int 15h for many other services. Today, int 15h is used for a wide variety of function including accessing expanded memory, reading the joystick / game adapter card, and many, many other operations. Except for the joystick calls, most of these services are beyond the scope of this text. Check on the bibliography if you interested in obtaining information on this BIOS call.

13.2.8 INT 16h - Keyboard Services

Instruction:	int 16h
BIOS Operation:	Read a key, test for a key, or get keyboard status
Parameters:	al

The IBM PC BIOS provides several function calls dealing with the keyboard. As with many of the PC BIOS routines, the number of functions has increased over the years. This section describes the three calls that were available on the original IBM PC.

1. For those who do not remember that far back, before there were hard disks people used to use only floppy disks. And before there were floppy disks, people used to use cassette tapes to store programs and data. The original IBM PC was introduced in late 1981 with a cassette port. By early 1982, no one was using cassette tape for data storage.

13.2.8.1 AH=0: Read a Key From the Keyboard

If int 16h is called with ah equal to zero, the BIOS will not return control to the caller until a key is available in the system type ahead buffer. On return, al contains the ASCII code for the key read from the buffer and ah contains the keyboard scan code. Keyboard scan codes are described in the appendices.

Certain keys on the PC's keyboard do not have any corresponding ASCII codes. The function keys, Home, PgUp, End, PgDn, the arrow keys, and the Alt keys are all good examples. When such a key is pressed, int 16h returns a zero in al and the keyboard scan code in ah. Therefore, whenever an ASCII code of zero is returned, you must check the ah register to determine which key was pressed.

Note that reading a key from the keyboard using the BIOS int 16h call does not echo the key pressed to the display. You have to call putc or use int 10h to print the character once you've read it if you want it echoed to the screen.

Example: Read a sequence of keystrokes from the keyboard until Enter is pressed.

```
ReadLoop:       mov     ah, 0       ;Read Key opcode
                int     16h
                cmp     al, 0       ;Special function?
                jz      ReadLoop    ;If so, don't echo this keystroke
                putc
                cmp     al, 0dh     ;Carriage return (ENTER)?
                jne     ReadLoop
```

13.2.8.2 AH=1: See if a Key is Available at the Keyboard

This particular int 16h subfunction allows you to check to see if a key is available in the system type ahead buffer. Even if a key is not available, control is returned (right away!) to the caller. With this call you can occasionally poll the keyboard to see if a key is available and continue processing if a key hasn't been pressed (as opposed to freezing up the computer until a key is pressed).

There are no input parameters to this function. On return, the zero flag will be clear if a key is available, set if there aren't any keys in the type ahead buffer. If a key is available, then ax will contain the scan and ASCII codes for that key. However, this function will not remove that keystroke from the typeahead buffer. Subfunction #0 must be used to remove characters. The following example demonstrates how to build a random number generator using the test keyboard function:

Example: Generating a random number while waiting for a keystroke

```
; First, clear any characters out of the type ahead buffer

ClrBuffer:      mov     ah, 1       ;Is a key available?
                int     16h
                jz      BufferIsClr ;If not, Discontinue flushing
                mov     ah, 0       ;Flush this character from the
                int     16h         ; buffer and try again.
                jmp     ClrBuffer

BufferIsClr:    mov     cx, 0       ;Initialize "random" number.
GenRandom:      inc     cx
                mov     ah, 1       ;See if a key is available yet.
                int     16h
                jz      GenRandom
                xor     cl, ch      ;Randomize even more.
                mov     ah, 0       ;Read character from buffer
                int     16h

; Random number is now in CL, key pressed by user is in AX
```

While waiting for a key, this routine is constantly incrementing the cx register. Since human beings cannot respond rapidly (at least in terms of microseconds) the cl register will overflow many times, even for the fastest typist. As a result, cl will contain a random value since the user will not be able to control (to better than about 2ms) when a key is pressed.

13.2.8.3 AH=2: Return Keyboard Shift Key Status

This function returns the state of various keys on the PC keyboard in the al register. The values returned are as follows:

Bit	Meaning
7	Insert state (toggle by pressing INS key)
6	Caps lock (1=capslock on)
5	Num lock (1=numlock on)
4	Scroll lock (1=scroll lock on)
3	Alt (1=Alt key currently down)
2	Ctrl (1=Ctrl key currently down)
1	Left shift (1=left shift key down)
0	Right shift (1=right shift key down)

Due to a bug in the BIOS code, these bits only reflect the current status of these keys, they do not necessarily reflect the status of these keys when the next key to be read from the system type ahead buffer was depressed. In order to ensure that these status bits correspond to the state of these keys when a scan code is read from the type ahead buffer, you've got to flush the buffer, wait until a key is pressed, and then immediately check the keyboard status.

13.2.9 INT 17h - Printer Services

Instruction:	int 17h
BIOS Operation:	Print data and test the printer status
Parameters:	ax, dx

Int 17h controls the parallel printer interfaces on the IBM PC in much the same way the int 14h controls the serial ports. Since programming a parallel port is considerably easier than controlling a serial port, using the int 17h routine is somewhat easier than using the int 14h routines.

Int 17h provides three subfunctions, specified by the value in the ah register. These subfunctions are:

0-Print the character in the AL register.
1-Initialize the printer.
2-Return the printer status.

Each of these functions is described in the following sections.

Like the serial port services, the printer port services allow you to specify which of the three printers installed in the system you wish to use (LPT1:, LPT2:, or LPT3:). The value in the dx register (0..2) specifies which printer port is to be used.

One final note- under DOS it's possible to redirect all printer output to a serial port. This is quite useful if you're using a serial printer. The BIOS printer services only talk to parallel printer adapters. If you need to send data to a serial printer using BIOS, you'll have to use int 14h to transmit the data through a serial port.

13.2.9.1 AH=0: Print a Character

If ah is zero when you call int 17h, then the BIOS will print the character in the al register. Exactly how the character code in the al register is treated is entirely up to the printer device you're using. Most printers, however, respect the printable ASCII character set and a few control characters as well. Many printers will also print all the symbols in the IBM/ASCII character set (including European, line drawing, and other special symbols). Most printers treat control characters (especially ESC sequences) in completely different manners. Therefore, if you intend to print something other than standard ASCII characters, be forewarned that your software may not work on printers other than the brand you're developing your software on.

Upon return from the int 17h subfunction zero routine, the ah register contains the current status. The values actually returned are described in the section on subfunction number two.

13.2.9.2 AH=1: Initialize Printer

Executing this call sends an electrical impulse to the printer telling it to initialize itself. On return, the ah register contains the printer status as per function number two.

13.2.9.3 AH=2: Return Printer Status

This function call checks the printer status and returns it in the ah register. The values returned are:

AH:	Bit Meaning
7	1=Printer busy, 0=printer not busy
6	1=Acknowledge from printer
5	1=Out of paper signal
4	1=Printer selected
3	1=I/O error
2	Not used
1	Not used
0	Time out error

Acknowledge from printer is, essentially, a redundant signal (since printer busy/not busy gives you the same information). As long as the printer is busy, it will not accept additional data. Therefore, calling the print character function (ah=0) will result in a delay.

The out of paper signal is asserted whenever the printer detects that it is out of paper. This signal is not implemented on many printer adapters. On such adapters it is always programmed to a logic zero (even if the printer is out of paper). Therefore, seeing a zero in this bit position doesn't always guarantee that there is paper in the machine. Seeing a one here, however, definitely means that your printer is out of paper.

The printer selected bit contains a one as long as the printer is on-line. If the user takes the printer off-line, then this bit will be cleared.

The I/O error bit contains a one if some general I/O error has occurred.

The time out error bit contains a one if the BIOS routine waited for an extended period of time for the printer to become "not busy" yet the printer remained busy.

Note that certain peripheral devices (other than printers) also interface to the parallel port, often in addition to a parallel printer. Some of these devices use the error/status signal lines to return data to the PC. The software controlling such devices often takes over the int 17h routine (via a technique we'll talk about later on) and always returns a "no error" status or "time out error" status if an error occurs on the printing device. Therefore,

you should take care not to depend too heavily on these signals changing when you make the int 17h BIOS calls.

13.2.10 INT 18h - Run BASIC

Instruction: int 18h
BIOS Operation: Activate ROM BASIC
Parameters: None

Executing int 18h activates the ROM BASIC interpreter in an IBM PC. However, you shouldn't use this mechanism to run BASIC since many PC compatibles do not have BASIC in ROM and the result of executing int 18h is undefined.

13.2.11 INT 19h - Reboot Computer

Instruction: int 19h
BIOS Operation: Restart the system
Parameters: None

Executing this interrupt has the same effect as pressing control-alt-del on the keyboard. For obvious reasons, this interrupt service should be handled carefully!

13.2.12 INT 1Ah - Real Time Clock

Instruction: int 1ah
BIOS Operation: Real time clock services
Parameters: ax, cx, dx

There are two services provided by this BIOS routine- read the clock and set the clock. The PC's real time clock maintains a counter that counts the number of 1/18ths of a second that have transpired since midnight. When you read the clock, you get the number of "ticks" which have occurred since then. When you set the clock, you specify the number of "ticks" which have occurred since midnight. As usual, the particular service is selected via the value in the ah register.

13.2.12.1 AH=0: Read the Real Time Clock

If ah = 0, then int 1ah returns a 33-bit value in al:cx:dx as follows:

Reg	Value Returned
dx	L.O. word of clock count
cx	H.O. word of clock count
al	Zero if timer has not run for more than 24 hours Non-zero otherwise.

The 32-bit value in cx:dx represents the number of 55 millisecond periods which have elapsed since midnight.

13.2.12.2 AH=1: Setting the Real Time Clock

This call allows you to set the current system time value. cx:dx contains the current count (in 55ms increments) since last midnight. Cx contains the H.O. word, dx contains the L.O. word.

13.3 An Introduction to MS-DOS™

MS-DOS provides all of the basic file manager and device manager functions required by most application programs running on an IBM PC. MS-DOS handles file I/O, character I/0, memory management, and other miscellaneous functions in a (relatively) consistent manner. If you're serious about writing software for the PC, you'll have to get real friendly with MS-DOS.

The title of this section is "An Introduction to MS-DOS". And that's exactly what it means. There is no way MS-DOS can be completely covered in a single chapter. Given all of the different books that already exist on the subject, it probably cannot even be covered by a single book (it certainly hasn't been yet. Microsoft wrote a 1,600 page book on the subject and it didn't even cover the subject fully). All this is leading up to a cop-out. There is no way this subject can be treated in more than a superficial manner in a single chapter. If you're serious about writing programs in assembly language for the PC, you'll need to complement this text with several others. Additional books on MS-DOS include: MS-DOS Programmer's Reference (also called the MS-DOS Technical Reference Manual), Peter Norton's Programmer's Guide to the IBM PC, The MS-DOS Encyclopedia, and the MS-DOS Developer's Guide. This, of course, is only a partial list of the books that are available. See the bibliography in the appendices for more details. Without a doubt, the MS-DOS Technical Reference Manual is the most important text to get your hands on. This is the official description of MS-DOS calls and parameters.

MS-DOS has a long and colorful history[2]. Throughout its lifetime, it has undergone several revisions, each purporting to be better than the last. MS-DOS' origins go all the way back to the CP/M-80 operating system written for the Intel 8080 microprocessor chip. In fact, MS-DOS v1.0 was nothing much more than a clone of CP/M-80 for Intel's 8088 microprocessor. Unfortunately, CP/M-80's file handling capabilities were horrible, to say the least. Therefore, DOS[3] improved on CP/M. New file handling capabilities, compatible with Xenix and Unix, were added to DOS, producing MS-DOS v2.0. Additional calls were added to later versions of MS-DOS. Even with the introduction of OS/2 and Windows NT (which, as this is being written, have yet to take the world by storm), Microsoft is still working on enhancements to MS-DOS which may produce even later versions.

Each new feature added to DOS introduced new DOS functions while preserving all of the functionality of the previous versions of DOS. When Microsoft rewrote the DOS file handling routines in version two, they didn't replace the old calls, they simply added new ones. While this preserved software compatibility of programs that ran under the old version of DOS, what it produced was a DOS with two sets of functionally identical, but otherwise incompatible, file services.

We're only going to concentrate on a small subset of the available DOS commands in this chapter. We're going to totally ignore those obsolete commands that have been augmented by newer, better, commands in later versions of DOS. Furthermore, we're going to skip over a description of those calls that have very little use in day to day programming. For a complete, detailed, look at the commands not covered in this chapter, you should consider the acquisition of one of the aforementioned books.

2. The MS-DOS Encyclopedia gives Microsoft's account of the history of MS-DOS. Of course, this is a one-sided presentation, but it's interesting nonetheless.
3. This text uses "DOS" to mean MS-DOS.

13.3.1 MS-DOS Calling Sequence

MS-DOS is called via the int 21h instruction. To select an appropriate DOS function, you load the ah register with a function number before issuing the int 21h instruction. Most DOS calls require other parameters as well. Generally, these other parameters are passed in the CPU's register set. The specific parameters will be discussed along with each call. Unless MS-DOS returns some specific value in a register, all of the CPU's registers are preserved across a call to DOS[4].

13.3.2 MS-DOS Character Oriented Functions

DOS provides 12 character oriented I/O calls. Most of these deal with writing and reading data to/from the keyboard, video display, serial port, and printer port. All of these functions have corresponding BIOS services. In fact, DOS usually calls the appropriate BIOS function to handle the I/O operation. However, due to DOS' redirected I/O and device driver facilities, these functions don't always call the BIOS routines. Therefore, you shouldn't call the BIOS routines (rather than DOS) simply because DOS ends up calling BIOS. Doing so may prevent your program from working with certain DOS-supported devices.

Except for function code seven, all of the following character oriented calls check the console input device (keyboard) for a control-C. If the user presses a control-C, DOS executes an int 23h instruction. Usually, this instruction will cause the program to abort and control will be returned to DOS. Keep this in mind when issuing these calls.

Microsoft considers these calls obsolete and does not guarantee they will be present in future versions of DOS. So take these first 12 routines with a rather large grain of salt. Note that the UCR Standard Library provides the functionality of many of these calls anyway, and they make the proper DOS calls, not the obsolete ones.

Table 51: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
1		al- char read	Console Input w/Echo: Reads a single character from the keyboard and displays typed character on screen.
2	dl- output char		Console Output: Writes a single character to the display.
3		al- char read	Auxiliary Input: Reads a single character from the serial port.
4	dl- output char		Auxiliary Output: Writes a single character to the output port
5	dl- output char		Printer Output: Writes a single character to the printer

4. So Microsoft claims. This may or may not be true across all versions of DOS.

Table 51: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
6	dl- output char (if not 0FFh)	al- char read (if input dl = 0FFh)	Direct Console I/O: On input, if dl contains 0FFh, this function attempts to read a character from the keyboard. If a character is available, it returns the zero flag clear and the character in al. If no character is available, it returns the zero flag set. On input, if dl contains a value other than 0FFh, this routine sends the character to the display. This routine does not do ctrl-C checking.
7		al- char read	Direct Console Input: Reads a character from the keyboard. Does not echo the character to the display. This call does not check for ctrl-C
8		al- char read	Read Keyboard w/o Echo: Just like function 7 above, except this call checks for ctrl-C.
9	ds:dx- pointer to string terminated with "$".		Display String: This function displays the characters from location ds:dx up to (but not including) a terminating "$" character.
0Ah	ds:dx- pointer to input buffer.		Buffered Keyboard Input: This function reads a line of text from the keyboard and stores it into the input buffer pointed at by ds:dx. The first byte of the buffer must contain a count between one and 255 that contains the maximum number of allowable characters in the input buffer. This routine stores the actual number of characters read in the second byte. The actual input characters begin at the third byte of the buffer.
0Bh		al- status (0=not ready, 0FFh=ready)	Check Keyboard Status: Determines whether a character is available from the keyboard.
0Ch	al- DOS opcode 0, 1, 6, 7, or 8	al- input character if opcode 1, 6, 7, or 8.	Flush Buffer: This call empties the system type ahead buffer and then executes the DOS command specified in the al register (if al=0, no further action).

Functions 1, 2, 3, 4, 5, 9, and 0Ah are obsolete and you should not use them. Use the DOS file I/O calls instead (opcodes 3Fh and 40h).

13.3.3 MS-DOS Drive Commands

MS-DOS provides several commands that let you set the default drive, determine which drive is the default, and perform some other operations. The following table lists those functions.

Table 52: DOS Disk Drive Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0Dh			Reset Drive: Flushes all file buffers to disk. Generally called by ctrl-C handlers or sections of code that need to guaranteed file consistency because an error may occur.
0Eh	dl- drive number	al- number of logical drives	Set Default Drive: sets the DOS default drive to the specified value (0=A, 1=B, 2=C, etc.). Returns the number of logical drives in the system, although they may not be contiguous from 0-al.
19H		al- default drive number	Get Default Drive: Returns the current system default drive number (0=A, 1=B, 2=C, etc.).
1Ah	ds:dx- Disk Transfer Area address.		Set Disk Transfer Area Address: Sets the address that MS-DOS uses for obsolete file I/O and Find First/Find Next commands.
1Bh		al- sectors/cluster cx- bytes/sector dx- # of clusters ds:bx - points at media descriptor byte	Get Default Drive Data: Returns information about the disk in the default drive. Also see function 36h. Typical values for the media descriptor byte include: 0F0h- 3.5” 0F8h- Hard disk 0F9h- 720K 3.5” or 1.2M 5.25” 0FAh- 320K 5.25” 0FBh- 640K 3.5” 0FCh- 180K 5.25” 0FDh- 360K 5.25: 0FEh- 160K 5.25” 0FFh- 320K 5.25”
1Ch	dl- drive number	See above	Get Drive Data: same as above except you can specify the drive number in the dl register (0=default, 1=A, 2=B, 3=C, etc.).

Table 52: DOS Disk Drive Functions

Function # (AH)	Input Parameters	Output Parameters	Description
1Fh		al- contains 0FFh if error, 0 if no error. ds:bx- ptr to DPB	Get Default Disk Parameter Block (DPB): If successful, this function returns a pointer to the following structure: Drive (byte) - Drive number (0-A, 1=B, etc.). Unit (byte) - Unit number for driver. SectorSize (word) - # bytes/sector. ClusterMask (byte) - sectors/cluster minus one. Cluster2 (byte) - $2^{clusters/sector}$ FirstFAT (word) - Address of sector where FAT starts. FATCount (byte) - # of FATs. RootEntries (word) - # of entries in root directory. FirstSector (word) - first sector of first cluster. MaxCluster (word) - # of clusters on drive, plus one. FATsize (word) - # of sectors for FAT. DirSector (word) - first sector containing directory. DriverAdrs (dword) - address of device driver. Media (byte) - media descriptor byte. FirstAccess (byte) - set if there has been an access to drive. NextDPB (dword) - link to next DPB in list. NextFree (word) - last allocated cluster. FreeCnt (word) - number of free clusters.
2Eh	al- verify flag (0=no verify, 1=verify on).		Set/Reset Verify Flag: Turns on and off write verification. Usually off since this is a slow operation, but you can turn it on when performing critical I/O.
2Fh		es:bx- pointer to DTA	Get Disk Transfer Area Address: Returns a pointer to the current DTA in es:bx..
32h	dl- drive number.	Same as 1Fh	Get DPB: Same as function 1Fh except you get to specify the driver number (0=default, 1=A, 2=B, 3=C, etc.).
33h	al- 05 (subfunction code)	dl- startup drive #.	Get Startup Drive: Returns the number of the drive used to boot DOS (1=A, 2=B, 3=C, etc.).
36h	dl- drive number.	ax- sectors/cluster bx- available clusters cx- bytes/sector dx- total clusters	Get Disk Free Space: Reports the amount of free space. This call supersedes calls 1Bh and 1Ch that only support drives up to 32Mbytes. This call handles larger drives. You can compute the amount of free space (in bytes) by bx*ax*cx. If an error occurs, this call returns 0FFFFh in ax.
54h		al- verify state.	Get Verify State: Returns the current state of the write verify flag (al=0 if off, al=1 if on).

13.3.4 MS-DOS "Obsolete" Filing Calls

DOS functions 0Fh - 18h, 1Eh, 20h-24h, and 26h - 29h are the functions left over from the days of CP/M-80. In general, you shouldn't bother at all with these calls since

MS-DOS v2.0 and later provides a much better way to accomplish the operations performed by these calls.

13.3.5 MS-DOS Date and Time Functions

The MS-DOS date and time functions return the current date and time based on internal values maintained by the real time clock (RTC). Functions provided by DOS include reading and setting the date and time. These date and time values are used to perform date and time stamping of files when files are created on the disk. Therefore, if you change the date or time, keep in mind that it will have an effect on the files you create thereafter. Note that the UCR Standard Library also provides a set of date and time functions which, in many cases, are somewhat easier to use than these DOS calls.

Table 53: Date and Time Functions

Function # (AH)	Input Parameters	Output Parameters	Description
2Ah		al- day (0=Sun, 1=Mon, etc.). cx- year dh- month (1=Jan, 2=Feb, etc.). dl- Day of month (1-31).	Get Date: returns the current MS-DOS date.
2Bh	cx- year (1980 - 2099) dh- month (1-12) dl- day (1-31)		Set Date: sets the current MS-DOS date.
2CH		ch- hour (24hr fmt) cl- minutes dh- seconds dl- hundredths	Get Time: reads the current MS-DOS time. Note that the hundredths of a second field has a resolution of $^1/_{18}$ second.
2Dh	ch- hour cl- minutes dh- seconds dl- hundredths		Set Time: sets the current MS-DOS time.

13.3.6 MS-DOS Memory Management Functions

MS-DOS provides three memory management functions- allocate, deallocate, and resize (modify). For most programs, these three memory allocation calls are not used. When DOS executes a program, it gives all of the available memory, from the start of that program to the end of RAM, to the executing process. Any attempt to allocate memory without first giving unused memory back to the system will produce an “insufficient memory” error.

Sophisticated programs which terminate and remain resident, run other programs, or perform complex memory management tasks, may require the use of these memory management functions. Generally these types of programs immediately deallocate all of the memory that they don’t use and then begin allocating and deallocating storage as they see fit.

Since these are complex functions, they shouldn't be used unless you have a very specific purpose for them. Misusing these commands may result in loss of system memory that can be reclaimed only by rebooting the system. Each of the following calls returns the error status in the carry flag. If the carry is clear on return, then the operation was completed successfully. If the carry flag is set when DOS returns, then the ax register contains one of the following error codes:

7- Memory control blocks destroyed
8- Insufficient memory
9- Invalid memory block address

Additional notes about these errors will be discussed as appropriate.

13.3.6.1 Allocate Memory

Function (ah): 48h
Entry parameters: bx- Requested block size (in paragraphs)
Exit parameters: If no error (carry clear):
ax:0 points at allocated memory block

If an error (carry set):
bx- maximum possible allocation size
ax- error code (7 or 8)

This call is used to allocate a block of memory. On entry into DOS, bx contains the size of the requested block in paragraphs (groups of 16 bytes). On exit, assuming no error, the ax register contains the segment address of the start of the allocated block. If an error occurs, the block is not allocated and the ax register is returned containing the error code. If the allocation request failed due to insufficient memory, the bx register is returned containing the maximum number of paragraphs actually available.

13.3.6.2 Deallocate Memory

Function (ah): 49h
Entry parameters: es:0- Segment address of block to be deallocated
Exit parameters: If the carry is set, ax contains the error code (7,9)

This call is used to deallocate memory allocated via function 48h above. The es register cannot contain an arbitrary memory address. It must contain a value returned by the allocate memory function. You cannot use this call to deallocate a portion of an allocated block. The modify allocation function is used for that operation.

13.3.6.3 Modify Memory Allocation

Function (ah): 4Ah
Entry parameters: es:0- address of block to modify allocation size
bx- size of new block
Exit parameters: If the carry is set, then
ax contains the error code 7, 8, or 9
bx contains the maximum size possible (if error 8)

This call is used to change the size of an allocated block. On entry, es must contain the segment address of the allocated block returned by the memory allocation function. Bx must contain the new size of this block in paragraphs. While you can almost always reduce the size of a block, you cannot normally increase the size of a block if other blocks have been allocated after the block being modified. Keep this in mind when using this function.

13.3.6.4 Advanced Memory Management Functions

The MS-DOS 58h opcode lets programmers adjust MS-DOS' memory allocation strategy and control the use of upper memory blocks (UMBs). There are four subfunctions to this call, with the subfunction value appearing in the al register. The following table describes these calls:

Table 54: Advanced Memory Management Functions

Function # (AH)	Input Parameters	Output Parameters	Description
58h	al-0	ax- strategy	Get Allocation Strategy: Returns the current allocation strategy in ax (see table below for details).
58h	al-1 bx- strategy		Set Allocation Strategy: Sets the MS-DOS allocation strategy to the value specified in bx (see the table below for details).
58H	al- 2	al- link flag	Get Upper Memory Link: Returns true/false (1/0) in al to determine whether a program can allocate memory in the upper memory blocks.
58h	al- 3 bx- link flag (0=no link, 1=link okay).		Set Upper Memory Link: Links or unlinks the upper memory area. When linked, an application can allocate memory from the UMB (using the normal DOS allocate call).

Table 55: Memory Allocation Strategies

Value	Name	Description
0	First Fit Low	Search conventional memory for the first free block of memory large enough to satisfy the allocation request. This is the default case.
1	Best Fit Low	Search conventional memory for the smallest block large enough to satisfy the request.
2	Last Fit Low	Search conventional memory from the highest address downward for the first block large enough to satisfy the request.
80h	First Fit High	Search high memory, then conventional memory, for the first available block that can satisfy the allocation request.
81h	Best Fit High	Search high memory, then conventional memory for the smallest block large enough to satisfy the allocation request.
82h	Last Fit High	Search high memory from high addresses to low, then conventional memory from high addresses to low, for the first block large enough to satisfy the request.
40h	First Fit Highonly	Search high memory only for the first block large enough to satisfy the request.
41h	Best Fit Highonly	Search high memory only for the smallest block large enough to satisfy the request.

Table 55: Memory Allocation Strategies

Value	Name	Description
42h	Last Fit Highonly	Search high memory only, from the end of memory downward, for the first block large enough to satisfy the request.

These different allocation strategies can have an impact on system performance. For an analysis of different memory management strategies, please consult a good operating systems theory text.

13.3.7 MS-DOS Process Control Functions

DOS provides several services dealing with loading, executing, and terminating programs. Many of these functions have been rendered obsolete by later versions of DOS. There are three[5] functions of general interest- program termination, terminate and stay resident, and execute a program. These three functions will be discussed in the following sections.

13.3.7.1 Terminate Program Execution

Function (ah): 4Ch
Entry parameters: al- return code
Exit parameters: Does not return to your program

This is the function call normally used to terminate your program. It returns control to the calling process (normally, but not necessarily, DOS). A return code can be passed to the calling process in the al register. Exactly what meaning this return code has is entirely up to you. This return code can be tested with the DOS "IF ERRORLEVEL return code" command in a DOS batch file. All files opened by the current process will be automatically closed upon program termination.

Note that the UCR Standard Library function "ExitPgm" is simply a macro which makes this particular DOS call. This is the normal way of returning control back to MS-DOS or some other program which ran the currently active application.

13.3.7.2 Terminate, but Stay Resident

Function (ah): 31h
Entry parameters: al- return code
dx- memory size, in paragraphs
Exit parameters: does not return to your program

This function also terminates program execution, but upon returning to DOS, the memory in use by the process is not returned to the DOS free memory pool. Essentially, the program remains in memory. Programs which remain resident in memory after returning to DOS are often called TSRs (terminate and stay resident programs).

When this command is executed, the dx register contains the number of memory paragraphs to leave around in memory. This value is measured from the beginning of the "program segment prefix", a segment marking the start of your file in memory. The address of the PSP (program segment prefix) is passed to your program in the ds register

5. Actually, there are others. See the DOS technical reference manual for more details. We will only consider these three here.

when your program is first executed. You'll have to save this value if your program is a TSR[6].

Programs that terminate and stay resident need to provide some mechanism for restarting. Once they return to DOS they cannot normally be restarted. Most TSRs patch into one of the interrupt vectors (such as a keyboard, printer, or serial interrupt vector) in order to restart whenever some hardware related event occurs (such as when a key is pressed). This is how "pop-up" programs like SmartKey work.

Generally, TSR programs are pop-ups or special device drivers. The TSR mechanism provides a convenient way for you to load your own routines to replace or augment BIOS' routines. Your program loads into memory, patches the appropriate interrupt vector so that it points at an interrupt handler internal to your code, and then terminates and stays resident. Now, when the appropriate interrupt instruction is executed, your code will be called rather than BIOS'.

There are far too many details concerning TSRs including compatibility issues, DOS re-entrancy issues, and how interrupts are processed, to be considered here. Additional details will appear in a later chapter.

13.3.7.3 Execute a Program

Function (ah): 40h
Entry parameters: ds:dx- pointer to pathname of program to execute
es:bx- Pointer to parameter block
al- 0=load and execute, 1=load only, 3=load overlay.
Exit parameters: If carry is set, ax contains one of the following error codes:
1- invalid function
2- file not found
5- access denied
8- not enough memory
10- bad environment
11- bad format

The execute (exec) function is an extremely complex, but at the same time, very useful operation. This command allows you to load or load and execute a program off of the disk drive. On entry into the exec function, the ds:dx registers contain a pointer to a zero terminated string containing the name of the file to be loaded or executed, es:bx points at a parameter block, and al contains zero or one depending upon whether you want to load and execute a program or simply load it into memory. On return, if the carry is clear, then DOS properly executed the command. If the carry flag is set, then DOS encountered an error while executing the command.

The filename parameter can be a full pathname including drive and subdirectory information. "B:\DIR1\DIR2\MYPGM.EXE" is a perfectly valid filename (remember, however, it must be zero terminated). The segmented address of this pathname is passed in the ds:dx registers.

The es:bx registers point at a parameter block for the exec call. This parameter block takes on three different forms depending upon whether a program is being loaded and executed (al=0), just loaded into memory (al=1), or loaded as an overlay (al=3).

If al=0, the exec call loads and executes a program. In this case the es:bx registers point at a parameter block containing the following values:

Offset	Description
0	A word value containing the segment address of the default environment (usually this is set to zero which implies the use of the standard DOS environment).
2	Double word pointer containing the segment address of a command line string.

6. DOS also provides a call which will return the PSP for your program.

6	Double word pointer to default FCB at address 5Ch
0Ah	Double word pointer to default FCB at address 6Ch

The environment area is a set of strings containing default pathnames and other information (this information is provided by DOS using the PATH, SET, and other DOS commands). If this parameter entry contains zero, then exec will pass the standard DOS environment on to the new procedure. If non-zero, then this parameter contains the segment address of the environment block that your process is passing on to the program about to be executed. Generally, you should store a zero at this address.

The pointer to the command string should contain the segmented address of a length prefixed string which is also terminated by a carriage return character (the carriage return character is not figured into the length of the string). This string corresponds to the data that is normally typed after the program name on the DOS command line. For example, if you're executing the linker automatically, you might pass a command string of the following form:

```
CmdStr      byte       16,"MyPgm+Routines /m",0dh
```

The second item in the parameter block must contain the segmented address of this string.

The third and fourth items in the parameter block point at the default FCBs. FCBs are used by the obsolete DOS filing commands, so they are rarely used in modern application programs. Since the data structures these two pointers point at are rarely used, you can point them at a group of 20 zeros.

Example: Format a floppy disk in drive A: using the FORMAT.EXE command

```
                mov     ah, 4Bh
                mov     al, 0
                mov     dx, seg PathName
                mov     ds, dx
                lea     dx, PathName
                mov     bx, seg ParmBlock
                mov     es, bx
                lea     bx, ParmBlock
                int     21h
                  .
                  .
                  .
PathName        byte    'C:\DOS\FORMAT.EXE',0
ParmBlock       word    0                       ;Default environment
                dword   CmdLine                 ;Command line string
                dword   Dummy,Dummy             ;Dummy FCBs

CmdLine         byte    3,' A:',0dh
Dummy           byte    20 dup (?)
```

MS-DOS versions earlier than 3.0 do not preserve any registers except cs:ip when you execute the exec call. In particular, ss:sp is not preserved. If you're using DOS v2.x or earlier, you'll need to use the following code:

```
;Example: Format a floppy disk in drive A: using the FORMAT.EXE command

                <push any registers you need preserved>

                mov     cs:SS_Save, ss          ;Save SS:SP to a location
                mov     cs:SP_Save, sp          ; we have access to later.
                mov     ah, 4Bh                 ;EXEC DOS opcode.
                mov     al, 0                   ;Load and execute.
                mov     dx, seg PathName        ;Get filename into DS:DX.
                mov     ds, dx
                lea     dx, PathName
                mov     bx, seg ParmBlock       ;Point ES:BX at parameter
                mov     es, bx                  ; block.
                lea     bx, ParmBlock
                int     21h
                mov     ss, cs:SS_Save          ;Restore SS:SP from saved
                mov     sp, cs:SP_Save          ; locations.
```

```
        <Restore registers pushed onto the stack>
                        .
                        .
                        .
SS_Save         word    ?
SP_Save         word    ?
                        .
                        .
                        .
PathName        byte    'C:\DOS\FORMAT.EXE',0
ParmBlock       word    0                       ;Default environment
                dword   CmdLine                 ;Command line string
                dword   Dummy,Dummy;Dummy       ;FCBs
CmdLine         byte    3,' A:',0dh
Dummy           byte    20 dup (?)
```

SS_Save and SP_Save must be declared inside your code segment. The other variables can be declared anywhere.

The exec command automatically allocates memory for the program being executed. If you haven't freed up unused memory before executing this command, you may get an insufficient memory error. Therefore, you should use the DOS deallocate memory command to free up unused memory before attempting to use the exec command.

If al=1 when the exec function executes, DOS will load the specified file but will not execute it. This function is generally used to load a program to execute into memory but give the caller control and let the caller start that code. When this function call is made, es:bx points at the following parameter block:

Offset	Description
0	Word value containing the segment address of the environment block for the new process. If you want to use the parent process' environment block set this word to zero.
2	Dword pointer to the command tail for this operation. The command tail is the command line string which will appear at location PSP:80 (See "The Program Segment Prefix (PSP)" on page 739 and "Accessing Command Line Parameters" on page 742).
6	Address of default FCB #1. For most programs, this should point at a block of 20 zeros (unless, of course, you're running a program which uses FCBs.).
0Ah	Address of default FCB #2. Should also point at a block of 20 zeros.
0Eh	SS:SP value. You must load these four bytes into SS and SP before starting the application.
12h	CS:IP value. These four bytes contain the starting address of the program.

The SSSP and CSIP fields are output values. DOS fills in the fields and returns them in the load structure. The other fields are all inputs which you must fill in before calling the exec function with al=1.

When you execute the exec command with al=-3, DOS simply loads an *overlay* into memory. Overlays generally consist of a single code segment which contains some functions you want to execute. Since you are not creating a new process, the parameter block for this type of load is much simpler than for the other two types of load operations. On entry, es:bx must point at the following parameter block in memory:

Offset	Description
0	Word value containing the segment address of where this file is going to be loaded into memory. The file will be loaded at offset zero within this segment.
2	Word value containing a relocation factor for this file.

Unlike the load and execute functions, the overlay function does not automatically allocate storage for the file being loaded. Your program has to allocate sufficient storage and then pass the address of this storage block to the exec command (though the parameter block above). Only the segment address of this block is passed to the exec command, the offset is always assumed to be zero. The relocation factor should also contain the segment address for ".EXE" files. For ".COM" files, the relocation factor parameter should be zero.

The overlay command is quite useful for loading overlays from disk into memory. An overlay is a segment of code which resides on the disk drive until the program actually needs to execute its code. Then the code is loaded into memory and executed. Overlays can reduce the amount of memory your program takes up by allowing you to reuse the same portion of memory for different overlay procedures (clearly, only one such procedure can be active at any one time). By placing seldom-used code and initialization code into overlay files, you can help reduce the amount of memory used by your program file. One word of caution, however, managing overlays is a very complex task. This is not something a beginning assembly language programmer would want to tackle right away. When loading a file into memory (as opposed to loading and executing a file), DOS does not scramble all of the registers, so you needn't take the extra care necessary to preserve the ss:sp and other registers.

The MS-DOS Encyclopedia contains an excellent description of the use of the exec function.

13.3.8 MS-DOS "New" Filing Calls

Starting with DOS v2.0, Microsoft introduced a set of file handling procedures which (finally) made disk file access bearable under MS-DOS. Not only bearable, but actually easy to use! The following sections describe the use of these commands to access files on a disk drive.

File commands which deal with filenames (Create, Open, Delete, Rename, and others) are passed the address of a zero-terminated pathname. Those that actually open a file (Create and Open) return a file handle as the result (assuming, of course, that there wasn't an error). This file handle is used with other calls (read, write, seek, close, etc.) to gain access to the file you've opened. In this respect, a file handle is not unlike a file variable in Pascal. Consider the following Microsoft/Turbo Pascal code:

```
program DemoFiles; var F:TEXT;
begin
        assign(f,'FileName.TXT');
        rewrite(f);
        writeln(f,'Hello there');
        close(f);
end.
```

The file variable "f" is used in this Pascal example in much the same way that a file handle is used in an assembly language program – to gain access to the file that was created in the program.

All the following DOS filing commands return an error status in the carry flag. If the carry flag is clear when DOS returns to your program, then the operation was completed successfully. If the carry flag is set upon return, then some sort of error has occurred and the AX register contains the error number. The actual error return values will be discussed along with each function in the following sections.

13.3.8.1 Open File

Function (ah): 3Dh
Entry parameters:
al- file access value
0- File opened for reading
1- File opened for writing
2- File opened for reading and writing
ds:dx- Point at a zero terminated string containing the filename.
Exit parameters: If the carry is set, ax contains one of the following error codes:
2- File not found

4- Too many open files
5- Access denied
12- Invalid access

If the carry is clear, ax contains the file handle value assigned by DOS.

A file must be opened before you can access it. The open command opens a file that already exists. This makes it quite similar to Pascal's Reset procedure. Attempting to open a file that doesn't exist produces an error. Example:

```
lea     dx, Filename            ;Assume DS points at segment
mov     ah, 3dh                 ; of filename
mov     al, 0                   ;Open for reading.
int     21h
jc      OpenError
mov     FileHandle, ax
```

If an error occurs while opening a file, the file will not be opened. You should always check for an error after executing a DOS open command, since continuing to operate on the file which hasn't been properly opened will produce disastrous consequences. Exactly how you handle an open error is up to you, but at the very least you should print an error message and give the user the opportunity to specify a different filename.

If the open command completes without generating an error, DOS returns a file handle for that file in the ax register. Typically, you should save this value away somewhere so you can use it when accessing the file later on.

13.3.8.2 Create File

Function (ah): 3Ch
Entry parameters: ds:dx- Address of zero terminated pathname
cx- File attribute
Exit parameters: If the carry is set, ax contains one of the following error codes:
3- Path not found
4- Too many open files
5- Access denied
If the carry is clear, ax is returned containing the file handle

Create opens a new file for output. As with the OPEN command, ds:dx points at a zero terminated string containing the filename. Since this call creates a new file, DOS assumes that you're opening the file for writing only. Another parameter, passed in cx, is the initial file attribute settings. The L.O. six bits of cx contain the following values:

Bit	Meaning if equal to one
0	File is a Read-Only file
1	File is a hidden file
2	File is a system file
3	File is a volume label name
4	File is a subdirectory
5	File has been archived

In general, you shouldn't set any of these bits. Most normal files should be created with a file attribute of zero. Therefore, the cx register should be loaded with zero before calling the create function.

Upon exit, the carry flag is set if an error occurs. The "Path not found" error requires some additional explanation. This error is generated, not if the file isn't found (which would be most of the time since this command is typically used to create a new file), but if a subdirectory in the pathname cannot be found.

If the carry flag is clear when DOS returns to your program, then the file has been properly opened for output and the ax register contains the file handle for this file.

13.3.8.3 Close File

Function (ah): 3Eh
Entry parameters: bx- File Handle
Exit parameters: If the carry flag is set, ax contains 6, the only possible error, which is an invalid handle error.

This call is used to close a file opened with the Open or Create commands above. It is passed the file handle in the bx register and, assuming the file handle is valid, closes the specified file.

You should close all files your program uses as soon as you're through with them to avoid disk file corruption in the event the user powers the system down or resets the machine while your files are left open.

Note that quitting to DOS (or aborting to DOS by pressing control-C or control-break) automatically closes all open files. However, you should never rely on this feature since doing so is an extremely poor programming practice.

13.3.8.4 Read From a File

Function (ah): 3Fh
Entry parameters: bx- File handle
cx- Number of bytes to read
ds:dx- Array large enough to hold bytes read
Exit parameters: If the carry flag is set, ax contains one of the following error codes
5- Access denied
6- Invalid handle
If the carry flag is clear, ax contains the number of bytes actually read from the file.

The read function is used to read some number of bytes from a file. The actual number of bytes is specified by the cx register upon entry into DOS. The file handle, which specifies the file from which the bytes are to be read, is passed in the bx register. The ds:dx register contains the address of a buffer into which the bytes read from the file are to be stored.

On return, if there wasn't an error, the ax register contains the number of bytes actually read. Unless the end of file (EOF) was reached, this number will match the value passed to DOS in the cx register. If the end of file has been reached, the value return in ax will be somewhere between zero and the value passed to DOS in the cx register. *This is the only test for the EOF condition.*

Example: This example opens a file and reads it to the EOF

```
                mov     ah, 3dh      ;Open the file
                mov     al, 0        ;Open for reading
                lea     dx, Filename ;Presume DS points at filename
                int     21h          ; segment.
                jc      BadOpen
                mov     FHndl, ax    ;Save file handle

LP:             mov     ah,3fh       ;Read data from the file
                lea     dx, Buffer   ;Address of data buffer
                mov     cx, 1        ;Read one byte
                mov     bx, FHndl    ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx       ;EOF reached?
                jne     EOF
                mov     al, Buffer   ;Get character read
                putc                 ;Print it
                jmp     LP           ;Read next byte

EOF:            mov     bx, FHndl
                mov     ah, 3eh      ;Close file
```

```
int     21h
jc      CloseError
```

This code segment will read the entire file whose (zero-terminated) filename is found at address "Filename" in the current data segment and write each character in the file to the standard output device using the UCR StdLib putc routine. Be forewarned that one-character-at-a-time I/O such as this is extremely slow. We'll discuss better ways to quickly read a file a little later in this chapter.

13.3.8.5 Write to a File

Function (ah): 40h
Entry parameters: bx- File handle
cx- Number of bytes to write
ds:dx- Address of buffer containing data to write
Exit parameters: If the carry is set, ax contains one of the following error codes
5- Accessed denied
6- Invalid handle
If the carry is clear on return, ax contains the number of bytes actually written to the file.

This call is almost the converse of the read command presented earlier. It writes the specified number of bytes at ds:dx to the file rather than reading them. On return, if the number of bytes written to the file is not equal to the number originally specified in the cx register, the disk is full and this should be treated as an error.

If cx contains zero when this function is called, DOS will truncate the file to the current file position (i.e., all data following the current position in the file will be deleted).

13.3.8.6 Seek (Move File Pointer)

Function (ah): 42h Entry parameters:
al- Method of moving
0- Offset specified is from the beginning of the file.
1- Offset specified is distance from the current file pointer.
2- The pointer is moved to the end of the file minus the specified offset.
bx- File handle.
cx:dx- Distance to move, in bytes.
Exit parameters: If the carry is set, ax contains one of the following error codes
1- Invalid function
6- Invalid handle
If the carry is clear, dx:ax contains the new file position

This command is used to move the file pointer around in a random access file. There are three methods of moving the file pointer, an absolute distance within the file (if al=0), some positive distance from the current file position (if al=1), or some distance from the end of the file (if al=2). If AL doesn't contain 0, 1, or 2, DOS will return an invalid function error. If this call is successfully completed, the next byte read or written will occur at the specified location.

Note that DOS treats cx:dx as an unsigned integer. Therefore, a single seek command cannot be used to move backwards in the file. Instead, method #0 must be used to position the file pointer at some absolute position in the file. If you don't know where you currently are and you want to move back 256 bytes, you can use the following code:

```
mov     ah, 42h             ;Seek command
mov     al, 1               ;Move from current location
xor     cx, cx              ;Zero out CX and DX so we
xor     dx, dx              ; stay right here
```

```
mov     bx, FileHandle
int     21h
jc      SeekError
sub     ax, 256                 ;DX:AX now contains the
sbb     dx, 0                   ; current file position, so
mov     cx, dx                  ; compute a location 256
mov     dx, ax                  ; bytes back.
mov     ah, 42h
mov     al, 0                   ;Absolute file position
int     21h                     ;BX still contains handle.
```

13.3.8.7 Set Disk Transfer Address (DTA)

Function (ah): 1Ah Entry parameters:
ds:dx- Pointer to DTA
Exit parameters: None

This command is called "Set Disk Transfer Address" because it was (is) used with the original DOS v1.0 file functions. We wouldn't normally consider this function except for the fact that it is also used by functions 4Eh and 4Fh (described next) to set up a pointer to a 43-byte buffer area. If this function isn't executed before executing functions 4Eh or 4Fh, DOS will use the default buffer space at PSP:80h.

13.3.8.8 Find First File

Function (ah): 4Eh
Entry parameters: cx- Attributes
ds:dx- Pointer to filename
Exit parameters: If carry is set, ax contains one of the following error codes
2- File not found
18- No more files

The Find First File and Find Next File (described next) functions are used to search for files specified using ambiguous file references. An ambiguous file reference is any filename containing the "*" and "?" wildcard characters. The Find First File function is used to locate the first such filename within a specified directory, the Find Next File function is used to find successive entries in the directory.

Generally, when an ambiguous file reference is provided, the Find First File command is issued to locate the first occurrence of the file, and then a loop is used, calling Find Next File, to locate all other occurrences of the file within that loop until there are no more files (error #18). Whenever Find First File is called, it sets up the following information at the DTA:

Offset	Description
0	Reserved for use by Find Next File
21	Attribute of file found
22	Time stamp of file
24	Date stamp of file
26	File size in bytes
30	Filename and extension (zero terminated)

(The offsets are decimal)

Assuming Find First File doesn't return some sort of error, the name of the first file matching the ambiguous file description will appear at offset 30 in the DTA.

Note: if the specified pathname doesn't contain any wildcard characters, then Find First File will return the exact filename specified, if it exists. Any subsequent call to Find Next File will return an error.

The cx register contains the search attributes for the file. Normally, cx should contain zero. If non-zero, Find First File (and Find Next File) will include file names which have the specified attributes as well as all normal file names.

13.3.8.9 Find Next File

Function (ah): 4Fh
Entry parameters: none
Exit parameters: If the carry is set, then there aren't any more files and ax will be returned containing 18.

The Find Next File function is used to search for additional file names matching an ambiguous file reference after a call to Find First File. The DTA must point at a data record set up by the Find First File function.

Example: The following code lists the names of all the files in the current directory that end with ".EXE". Presumably, the variable "DTA" is in the current data segment:

```
                mov     ah, 1Ah                 ;Set DTA
                lea     dx, DTA
                int     21h
                xor     cx, cx                  ;No attributes.
                lea     dx, FileName
                mov     ah, 4Eh                 ;Find First File
                int     21h
                jc      NoMoreFiles             ;If error, we're done
DirLoop:        lea     si, DTA+30              ;Address of filename
                cld
PrtName:        lodsb
                test    al, al                  ;Zero byte?
                jz      NextEntry
                putc                            ;Print this character
                jmp     PrtName

NextEntry:      mov     ah, 4Fh                 ;Find Next File
                int     21h
                jnc     DirLoop                 ;Print this name
```

13.3.8.10 Delete File

Function (ah): 41h
Entry parameters: ds:dx- Address of pathname to delete
Exit parameters: If carry set, ax contains one of the following error codes
2- File not found
5- Access denied

This function will delete the specified file from the directory. The filename must be an unambiguous filename (i.e., it cannot contain any wildcard characters).

13.3.8.11 Rename File

Function (ah): 56h Entry parameters:
ds:dx- Pointer to pathname of existing file
es:di- Pointer to new pathname
Exit parameters: If carry set, ax contains one of the following error codes
2- File not found
5- Access denied
17- Not the same device

This command serves two purposes: it allows you to rename one file to another and it allows you to move a file from one directory to another (as long as the two subdirectories are on the same disk).

Example: Rename “MYPGM.EXE” to “YOURPGM.EXE”

```
; Assume ES and DS both point at the current data segment
; containing the filenames.

                lea     dx, OldName
                lea     di, NewName
                mov     ah, 56h
                int     21h
                jc      BadRename
                        .
                        .
                        .
OldName         byte    “MYPGM.EXE”,0
NewName         byte    “YOURPGM.EXE”,0

Example #2: Move a filename from one directory to another:
; Assume ES and DS both point at the current data segment
; containing the filenames.

                lea     dx, OldName
                lea     di, NewName
                mov     ah, 56h
                int     21h
                jc      BadRename
                        .
                        .
                        .
OldName         byte    “\DIR1\MYPGM.EXE”,0
NewName         byte    “\DIR2\MYPGM.EXE”,0
```

13.3.8.12 Change/Get File Attributes

Function (ah): 43h
Entry parameters: al- Subfunction code
0- Return file attributes in cx
1- Set file attributes to those in cx
cx- Attribute to be set if AL=01
ds:dx- address of pathname
Exit parameters: If carry set, ax contains one of the following error codes:
1- Invalid function
3- Pathname not found
5- Access denied
If the carry is clear and the subfunction was zero cx will contain the file’s attributes.

This call is useful for setting/resetting and reading a file’s attribute bits. It can be used to set a file to read-only, set/clear the archive bit, or otherwise mess around with the file attributes.

13.3.8.13 Get/Set File Date and Time

Function (ah): 57h
Entry parameters: al- Subfunction code
0- Get date and time
1- Set date and time
bx- File handle
cx- Time to be set (if AL=01)
dx- Date to be set (if AL=01)

Exit parameters: If carry set, ax contains one of the following error codes
1- Invalid subfunction
6- Invalid handle
If the carry is clear, cx/dx is set to the time/date if al=00

This call sets the "last-write" date/time for the specified file. The file must be open (using open or create) before using this function. The date will not be recorded until the file is closed.

13.3.8.14 Other DOS Calls

The following tables briefly list many of the other DOS calls. For more information on the use of these DOS functions consult the Microsoft MS-DOS Programmer's Reference or the MS-DOS Technical Reference.

Table 56: Miscellaneous DOS File Functions

Function # (AH)	Input Parameters	Output Parameters	Description
39h	ds:dx- pointer to zero terminated pathname.		Create Directory: Creates a new directory with the specified name.
3Ah	ds:dx- pointer to zero terminated pathname.		Remove Directory: Deletes the directory with the specified pathname. Error if directory is not empty or the specified directory is the current directory.
3Bh	ds:dx- pointer to zero terminated pathname.		Change Directory: Changes the default directory to the specified pathname.
45h	bx- file handle	ax- new handle	Duplicate File Handle: creates a copy of a file handle so a program can access a file using two separate file variables. This allows the program to close the file with one handle and continue accessing it with the other.
46h	bx- file handle cx- duplicate handle		Force Duplicate File Handle: Like function 45h above, except you specify which handle (in cx) you want to refer to the existing file (specified by bx).
47h	ds:si- pointer to buffer dl- drive		Get Current Directory: Stores a string containing the current pathname (terminated with a zero) starting at location ds:si. These registers must point at a buffer containing at least 64 bytes. The dl register specifies the drive number (0=default, 1=A, 2=B, 3=C, etc.).
5Ah	cx- attributes ds:dx- pointer to temporary path.	ax- handle	Create Temporary File: Creates a file with a unique name in the directory specified by the zero terminated string at which ds:dx points. There must be at least 13 zero bytes beyond the end of the pathname because this function will store the generated filename at the end of the pathname. The attributes are the same as for the Create call.

Table 56: Miscellaneous DOS File Functions

Function # (AH)	Input Parameters	Output Parameters	Description
5Bh	cx- attributes ds:dx- pointer to zero terminated pathname.	ax- handle	Create New File: Like the create call, but this call insists that the file not exist. It returns an error if the file exists (rather than deleting the old file).
67h	bx- handles		Set Maximum Handle Count: This function sets the maximum number of handles a program can use at any one given time.
68h	bx- handle		Commit File: Flushes all data to a file without closing it, ensuring that the file's data is current and consistent.

Table 57: Miscellaneous DOS Functions

Function # (AH)	Input Parameters	Output Parameters	Description
25h	al- interrupt # ds:dx- pointer to interrupt service routine.		Set Interrupt Vector: Stores the specified address in ds:dx into the interrupt vector table at the entry specified by the al register.
30h		al- major version ah- minor version bh- Version flag bl:cx- 24 bit serial number	Get Version Number: Returns the current version number of DOS (or value set by SETVER).
33h	al- 0	dl- break flag (0=off, 1=on)	Get Break Flag: Returns the status of the DOS break flag. If on, MS-DOS checks for ctrl-C when processing any DOS command; if off, MS-DOS only checks on functions 1-0Ch.
33h	al- 1 dl- break flag.		Set Break Flag: Sets the MS-DOS break flag according to the value in dl (see function above for details).
33h	al- 6	bl- major version bh- minor version dl- revision dh- version flags	Get MS-DOS Version: Returns the "real" version number, not the one set by the SETVER command. Bits three and four of the version flags are one if DOS is in ROM or DOS is in high memory, respectively.
34h		es:bx- pointer to InDOS flag.	Get InDOS Flag Address: Returns the address of the InDOS flag. This flag helps prevent reentrancy in TSR applications
35h	al- interrupt #	es:bx- pointer to interrupt service routine.	Get Interrupt Vector: Returns a pointer to the interrupt service routine for the specified interrupt number. See function 25h above for more details.
44h	al- subcode Other parameters!		Device Control: This is a whole family of additional DOS commands to control various devices. See the DOS programmer's reference manual for more details.

Table 57: Miscellaneous DOS Functions

Function # (AH)	Input Parameters	Output Parameters	Description
4Dh		al- return value ah- termination method	Get Child Program Return Value: Returns the last result code from a child program in al. The ah register contains the termination method, which is one of the following values: 0-normal, 1-ctrl-C, 2-critical device error, 3-terminate and stay resident.
50h	bx- PSP address		Set PSP Address: Set DOS' current PSP address to the value specified in the bx register.
51h		bx- PSP address	Get PSP Address: Returns a pointer to the current PSP in the bx register.
59h		ax- extended error bh- error class bl- error action ch- error location	Get Extended Error: Returns additional information when an error occurs on a DOS call. See the DOS programmer's guide for more details on these errors and how to handle them.
5Dh	al- 0Ah ds:si- pointer to extended error structure.		Set Extended Error: copies the data from the extended error structure to DOS' internal record.

In addition to the above commands, there are several additional DOS calls that deal with networks and international character sets. See the MS-DOS reference for more details.

13.3.9 File I/O Examples

Of course, one of the main reasons for making calls to DOS is to manipulate files on a mass storage device. The following examples demonstrate some uses of character I/O using DOS.

13.3.9.1 Example #1: A Hex Dump Utility

This program dumps a file in hexadecimal format. The filename must be hard coded into the file (see "Accessing Command Line Parameters" later in this chapter).

```
                include     stdlib.a
                includelib  stdlib.lib

cseg            segment   byte public 'CODE'
                assume    cs:cseg, ds:dseg, es:dseg, ss:sseg

MainPgm         proc      far

; Properly set up the segment registers:

                mov       ax, seg dseg
                mov       ds, ax
                mov       es, ax
                mov       ah, 3dh
                mov       al, 0                 ;Open file for reading
                lea       dx, Filename          ;File to open
                int       21h
                jnc       GoodOpen
```

```
                print
                byte    'Cannot open file, aborting program...',cr,0
                jmp     PgmExit

GoodOpen:       mov     FileHandle, ax          ;Save file handle
                mov     Position, 0             ;Initialize file pos counter
ReadFileLp:     mov     al, byte ptr Position
                and     al, 0Fh                 ;Compute (Position MOD 16)
                jnz     NotNewLn                ;Start new line each 16 bytes
                putcr
                mov     ax, Position            ;Print offset into file
                xchg    al, ah
                puth
                xchg    al, ah
                puth
                print
                byte    ': ',0

NotNewLn:       inc     Position                ;Increment character count
                mov     bx, FileHandle
                mov     cx, 1                   ;Read one byte
                lea     dx, buffer              ;Place to store that byte
                mov     ah, 3Fh                 ;Read operation
                int     21h
                jc      BadRead
                cmp     ax, 1                   ;Reached EOF?
                jnz     AtEOF
                mov     al, Buffer              ;Get the character read and
                puth                            ; print it in hex
                mov     al, ' '                 ;Print a space between values
                putc
                jmp     ReadFileLp

BadRead:        print
                byte    cr, lf
                byte    'Error reading data from file, aborting'
                byte    cr,lf,0

AtEOF:          mov     bx, FileHandle          ;Close the file
                mov     ah, 3Eh
                int     21h

PgmExit:        ExitPgm
MainPgm         endp

cseg            ends
dseg            segment    byte public 'data'

Filename        byte       'hexdump.asm',0             ;Filename to dump
FileHandle      word       ?
Buffer          byte       ?
Position        word       0

dseg            ends

sseg            segment    byte stack 'stack'
stk             word       0ffh dup (?)
sseg            ends

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        MainPgm
```

13.3.9.2 Example #2: Upper Case Conversion

The following program reads one file, converts all the lower case characters to upper case, and writes the data to a second output file.

```
                include     stdlib.a
                includelib  stdlib.lib
```

```
cseg            segment     byte public 'CODE'
                assume      cs:cseg, ds:dseg, es:dseg, ss:sseg

MainPgm         proc        far

; Properly set up the segment registers:

                mov     ax, seg dseg
                mov     ds, ax
                mov     es, ax

;-----------------------------------------------------------------
;
; Convert UCCONVRT.ASM to uppercase
;
; Open input file:

                mov     ah, 3dh
                mov     al, 0                   ;Open file for reading
                lea     dx, Filename            ;File to open
                int     21h
                jnc     GoodOpen
                print
                byte    'Cannot open file, aborting program...',cr,lf,0
                jmp     PgmExit

GoodOpen:       mov     FileHandle1, ax         ;Save input file handle

; Open output file:

                mov     ah, 3Ch                 ;Create file call
                mov     cx, 0                   ;Normal file attributes
                lea     dx, OutFileName         ;File to open
                int     21h
                jnc     GoodOpen2
                print
                byte    'Cannot open output file, aborting program...'
                byte    cr,lf,0
                mov     ah, 3eh                 ;Close input file
                mov     bx, FileHandle1
                int     21h
                jmp     PgmExit                 ;Ignore any error.

GoodOpen2:      mov     FileHandle2, ax         ;Save output file handle

ReadFileLp:     mov     bx, FileHandle1
                mov     cx, 1                   ;Read one byte
                lea     dx, buffer              ;Place to store that byte
                mov     ah, 3Fh                 ;Read operation
                int     21h
                jc      BadRead
                cmp     ax, 1                   ;Reached EOF?
                jz      ReadOK
                jmp     AtEOF

ReadOK:         mov     al, Buffer              ;Get the character read and
                cmp     al, 'a'                 ; convert it to upper case
                jb      NotLower
                cmp     al, 'z'
                ja      NotLower
                and     al, 5fh                 ;Set Bit #5 to zero.
NotLower:       mov     Buffer, al

; Now write the data to the output file

                mov     bx, FileHandle2
                mov     cx, 1                   ;Read one byte
                lea     dx, buffer              ;Place to store that byte
                mov     ah, 40h                 ;Write operation
                int     21h
                jc      BadWrite
                cmp     ax, 1                   ;Make sure disk isn't full
                jz      ReadFileLp

BadWrite:       print
```

```
                byte    cr, lf
                byte    'Error writing data to file, aborting operation'
                byte    cr,lf,0
                jmp     short AtEOF

BadRead:        print
                byte    cr, lf
                byte    'Error reading data from file, aborting '
                byte    'operation',cr,lf,0

AtEOF:          mov     bx, FileHandle1         ;Close the file
                mov     ah, 3Eh
                int     21h
                mov     bx, FileHandle2
                mov     ah, 3eh
                int     21h

;-----------------------------------------------------------------

PgmExit:        ExitPgm
MainPgm         endp
cseg            ends

dseg            segment byte public 'data'

Filename        byte      'ucconvrt.asm',0            ;Filename to convert
OutFileName     byte      'output.txt',0              ;Output filename
FileHandle1     word      ?
FileHandle2     word      ?
Buffer          byte      ?
Position        word      0

dseg            ends

sseg            segment   byte stack 'stack'
stk             word      0ffh dup (?)
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end     MainPgm
```

13.3.10 Blocked File I/O

The examples in the previous section suffer from a major drawback, they are extremely slow. The performance problems with the code above are entirely due to DOS. Making a DOS call is not, shall we say, the fastest operation in the world. Calling DOS every time we want to read or write a single character from/to a file will bring the system to its knees. As it turns out, it doesn't take (practically) any more time to have DOS read or write two characters than it does to read or write one character. Since the amount of time we (usually) spend processing the data is negligible compared to the amount of time DOS takes to return or write the data, reading two characters at a time will essentially double the speed of the program. If reading two characters doubles the processing speed, how about reading four characters? Sure enough, it almost quadruples the processing speed. Likewise processing ten characters at a time almost increases the processing speed by an order of magnitude. Alas, this progression doesn't continue forever. There comes a point of diminishing returns- when it takes far too much memory to justify a (very) small improvement in performance (keeping in mind that reading 64K in a single operation requires a 64K memory buffer to hold the data). A good compromise is 256 or 512 bytes. Reading more data doesn't really improve the performance much, yet a 256 or 512 byte buffer is easier to deal with than larger buffers.

Reading data in groups or blocks is called *blocked I/O*. Blocked I/O is often one to two orders of magnitude faster than single character I/O, so obviously you should use blocked I/O whenever possible.

There is one minor drawback to blocked I/O-- it's a little more complex to program than single character I/O. Consider the example presented in the section on the DOS read command:

Example: This example opens a file and reads it to the EOF

```
                mov     ah, 3dh                 ;Open the file
                mov     al, 0                   ;Open for reading
                lea     dx, Filename            ;Presume DS points at
filename
                int     21h                     ; segment
                jc      BadOpen
                mov     FHndl, ax               ;Save file handle

LP:             mov     ah,3fh                  ;Read data from the file
                lea     dx, Buffer              ;Address of data buffer
                mov     cx, 1                   ;Read one byte
                mov     bx, FHndl               ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx                  ;EOF reached?
                jne     EOF
                mov     al, Buffer              ;Get character read
                putc                            ;Print it (IOSHELL call)
                jmp     LP                      ;Read next byte

EOF:            mov     bx, FHndl
                mov     ah, 3eh                 ;Close file
                int     21h
                jc      CloseError
```

There isn't much to this program at all. Now consider the same example rewritten to use blocked I/O:

Example: This example opens a file and reads it to the EOF using blocked I/O

```
                mov     ah, 3dh                 ;Open the file
                mov     al, 0                   ;Open for reading
                lea     dx, Filename            ;Presume DS points at
filename
                int     21h                     ; segment
                jc      BadOpen
                mov     FHndl, ax               ;Save file handle

LP:             mov     ah,3fh                  ;Read data from the file
                lea     dx, Buffer              ;Address of data buffer
                mov     cx, 256                 ;Read 256 bytes
                mov     bx, FHndl               ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx;EOF reached?
                jne     EOF
                mov     si, 0                   ;Note: CX=256 at this point.
PrtLp:          mov     al, Buffer[si]          ;Get character read
                putc                            ;Print it
                inc     si
                loop    PrtLp
                jmp     LP                      ;Read next block

; Note, just because the number of bytes read doesn't equal 256,
; don't get the idea we're through, there could be up to 255 bytes
; in the buffer still waiting to be processed.

EOF:            mov     cx, ax
                jcxz    EOF2            ;If CX is zero, we're really done.
                mov     si, 0           ;Process the last block of data read
Finis:          mov     al, Buffer[si]; from the file which contains
                putc                    ; 1..255 bytes of valid data.
                inc     si
                loop    Finis

EOF2:           mov     bx, FHndl
                mov     ah, 3eh ;Close file
```

```
                int     21h
                jc      CloseError
```

This example demonstrates one major hassle with blocked I/O – when you reach the end of file, you haven't necessarily processed all of the data in the file. If the block size is 256 and there are 255 bytes left in the file, DOS will return an EOF condition (the number of bytes read don't match the request). In this case, we've still got to process the characters that were read. The code above does this in a rather straight-forward manner, using a second loop to finish up when the EOF is reached. You've probably noticed that the two print loops are virtually identical. This program can be reduced in size somewhat using the following code which is only a little more complex:

Example: This example opens a file and reads it to the EOF using blocked I/O

```
                mov     ah, 3dh                 ;Open the file
                mov     al, 0                   ;Open for reading
                lea     dx, Filename            ;Presume DS points at
filename
                int     21h                     ; segment.
                jc      BadOpen
                mov     FHndl, ax               ;Save file handle

LP:             mov     ah,3fh                  ;Read data from the file
                lea     dx, Buffer              ;Address of data buffer
                mov     cx, 256                 ;Read 256 bytes
                mov     bx, FHndl               ;Get file handle value
                int     21h
                jc      ReadError
                mov     bx, ax                  ;Save for later
                mov     cx, ax
                jcxz    EOF
                mov     si, 0                   ;Note: CX=256 at this point.
PrtLp:          mov     al, Buffer[si]          ;Get character read
                putc                            ;Print it
                inc     si
                loop    PrtLp
                cmp     bx, 256                 ;Reach EOF yet?
                je      LP

EOF:            mov     bx, FHndl
                mov     ah, 3eh                 ;Close file
                int     21h
                jc      CloseError
```

Blocked I/O works best on sequential files. That is, those files opened only for reading or writing (no seeking). When dealing with random access files, you should read or write whole records at one time using the DOS read/write commands to process the whole record. This is still considerably faster than manipulating the data one byte at a time.

13.3.11 The Program Segment Prefix (PSP)

When a program is loaded into memory for execution, DOS first builds up a program segment prefix immediately before the program is loaded into memory. This PSP contains lots of information, some of it useful, some of it obsolete. Understanding the layout of the PSP is essential for programmers designing assembly language programs.

The PSP is 256 bytes long and contains the following information:

Offset	Length	Description
0	2	An INT 20h instruction is stored here
2	2	Program ending address
4	1	Unused, reserved by DOS
5	5	Call to DOS function dispatcher
0Ah	4	Address of program termination code

0Eh	4	Address of break handler routine
12h	4	Address of critical error handler routine
16h	22	Reserved for use by DOS
2Ch	2	Segment address of environment area
2Eh	34	Reserved by DOS
50h	3	INT 21h, RETF instructions
53h	9	Reserved by DOS
5Ch	16	Default FCB #1
6Ch	20	Default FCB #2
80h	1	Length of command line string
81h	127	Command line string

Note: locations 80h..FFh are used for the default DTA.

Most of the information in the PSP is of little use to a modern MS-DOS assembly language program. Buried in the PSP, however, are a couple of gems that are worth knowing about. Just for completeness, however, we'll take a look at all of the fields in the PSP.

The first field in the PSP contains an int 20h instruction. Int 20h is an obsolete mechanism used to terminate program execution. Back in the early days of DOS v1.0, your program would execute a jmp to this location in order to terminate. Nowadays, of course, we have DOS function 4Ch which is much easier (and safer) than jumping to location zero in the PSP. Therefore, this field is obsolete.

Field number two contains a value which points at the last paragraph allocated to your program By subtracting the address of the PSP from this value, you can determine the amount of memory allocated to your program (and quit if there is insufficient memory available).

The third field is the first of many "holes" left in the PSP by Microsoft. Why they're here is anyone's guess.

The fourth field is a call to the DOS function dispatcher. The purpose of this (now obsolete) DOS calling mechanism was to allow some additional compatibility with CP/M-80 programs. For modern DOS programs, there is absolutely no need to worry about this field.

The next three fields are used to store special addresses during the execution of a program. These fields contain the default terminate vector, break vector, and critical error handler vectors. These are the values normally stored in the interrupt vectors for int 22h, int 23h, and int 24h. By storing a copy of the values in the vectors for these interrupts, you can change these vectors so that they point into your own code. When your program terminates, DOS restores those three vectors from these three fields in the PSP. For more details on these interrupt vectors, please consult the DOS technical reference manual.

The eighth field in the PSP record is another reserved field, currently unavailable for use by your programs.

The ninth field is another real gem. It's the address of the environment strings area. This is a two-byte pointer which contains the segment address of the environment storage area. The environment strings always begin with an offset zero within this segment. The environment string area consists of a sequence of zero-terminated strings. It uses the following format:

$string_1$ 0 $string_2$ 0 $string_3$ 0 ... 0 $string_n$ 0 0

That is, the environment area consists of a list of zero terminated strings, the list itself being terminated by a string of length zero (i.e., a zero all by itself, or two zeros in a row, however you want to look at it). Strings are (usually) placed in the environment area via DOS commands like PATH, SET, etc. Generally, a string in the environment area takes the form

```
name = parameters
```

For example, the "SET IPATH=C:\ASSEMBLY\INCLUDE" command copies the string "IPATH=C:\ASSEMBLY\INCLUDE" into the environment string storage area.

Many languages scan the environment storage area to find default filename paths and other pieces of default information set up by DOS. Your programs can take advantage of this as well.

The next field in the PSP is another block of reserved storage, currently undefined by DOS.

The 11th field in the PSP is another call to the DOS function dispatcher. Why this call exists (when the one at location 5 in the PSP already exists and nobody really uses either mechanism to call DOS) is an interesting question. In general, this field should be ignored by your programs.

The 12th field is another block of unused bytes in the PSP which should be ignored.

The 13th and 14th fields in the PSP are the default FCBs (File Control Blocks). File control blocks are another archaic data structure carried over from CP/M-80. FCBs are used only with the obsolete DOS v1.0 file handling routines, so they are of little interest to us. We'll ignore these FCBs in the PSP.

Locations 80h through the end of the PSP contain a very important piece of information- the command line parameters typed on the DOS command line along with your program's name. If the following is typed on the DOS command line:

```
MYPGM parameter1, parameter2
```

the following is stored into the command line parameter field:

```
23, " parameter1, parameter2", 0Dh
```

Location 80h contains 23_{10}, the length of the parameters following the program name. Locations 81h through 97h contain the characters making up the parameter string. Location 98h contains a carriage return. Notice that the carriage return character is not figured into the length of the command line string.

Processing the command line string is such an important facet of assembly language programming that this process will be discussed in detail in the next section.

Locations 80h..FFh in the PSP also comprise the default DTA. Therefore, if you don't use DOS function 1Ah to change the DTA and you execute a FIND FIRST FILE, the filename information will be stored starting at location 80h in the PSP.

One important detail we've omitted until now is exactly how you access data in the PSP. Although the PSP is loaded into memory immediately before your program, that doesn't necessarily mean that it appears 100h bytes before your code. Your data segments may have been loaded into memory before your code segments, thereby invalidating this method of locating the PSP. The segment address of the PSP is passed to your program in the ds register. To store the PSP address away in your data segment, your programs should begin with the following code:

```
push    ds                      ;Save PSP value
mov     ax, seg DSEG            ;Point DS and ES at our data
mov     ds, ax                  ; segment.
mov     es, ax
pop     PSP                     ;Store PSP value into "PSP"
                                ; variable.
  .
  .
  .
```

Another way to obtain the PSP address, in DOS 5.0 and later, is to make a DOS call. If you load ah with 51h and execute an int 21h instruction, MS-DOS will return the segment address of the current PSP in the bx register.

There are lots of tricky things you can do with the data in the PSP. Peter Norton's Programmer's Guide to the IBM PC lists all kinds of tricks. Such operations won't be discussed here because they're a little beyond the scope of this manual.

13.3.12 Accessing Command Line Parameters

Most programs like MASM and LINK allow you to specify command line parameters when the program is executed. For example, by typing

```
ML MYPGM.ASM
```

you can instruct MASM to assemble MYPGM without any further intervention from the keyboard. “MYPGM.ASM;” is a good example of a command line parameter.

When DOS’ COMMAND.COM command interpreter parses your command line, it copies most of the text following the program name to location 80h in the PSP as described in the previous section. For example, the command line above will store the following at PSP:80h

```
11, “ MYPGM.ASM”, 0Dh
```

The text stored in the command line tail storage area in the PSP is usually an exact copy of the data appearing on the command line. There are, however, a couple of exceptions. First of all, I/O redirection parameters are not stored in the input buffer. Neither are command tails following the pipe operator (“|”). The other thing appearing on the command line which is absent from the data at PSP:80h is the program name. This is rather unfortunate, since having the program name available would allow you to determine the directory containing the program. Nevertheless, there is lots of useful information present on the command line.

The information on the command line can be used for almost any purpose you see fit. However, most programs expect two types of parameters in the command line parameter buffer-- filenames and switches. The purpose of a filename is rather obvious, it allows a program to access a file without having to prompt the user for the filename. Switches, on the other hand, are arbitrary parameters to the program. By convention, switches are preceded by a slash or hyphen on the command line.

Figuring out what to do with the information on the command line is called *parsing* the command line. Clearly, if your programs are to manipulate data on the command line, you’ve got to parse the command line within your code.

Before a command line can be parsed, each item on the command line has to be separated out apart from the others. That is, each word (or more properly, lexeme[7]) has to be identified in the command line. Separation of lexemes on a command line is relatively easy, all you’ve got to do is look for sequences of delimiters on the command line. Delimiters are special symbols used to separate tokens on the command line. DOS supports six different delimiter characters: space, comma, semicolon, equal sign, tab, or carriage return.

Generally, any number of delimiter characters may appear between two tokens on a command line. Therefore, all such occurrences must be skipped when scanning the command line. The following assembly language code scans the entire command line and prints all of the tokens that appear thereon:

```
                include     stdlib.a
                includelib  stdlib.lib

cseg            segment     byte public ‘CODE’
                assume      cs:cseg, ds:dseg, es:dseg, ss:sseg

; Equates into command line-

CmdLnLen        equ      byte ptr es:[80h]      ;Command line length
CmdLn           equ      byte ptr es:[81h]      ;Command line data

tab             equ      09h

MainPgm         proc     far

; Properly set up the segment registers:
```

7. Many programmers use the term “token” rather than lexeme. Technically, a token is a different entity.

```
                push    ds                      ;Save PSP
                mov     ax, seg dseg
                mov     ds, ax
                pop     PSP

;-----------------------------------------------------------------

                print
                byte    cr,lf
                byte    'Items on this line:',cr,lf,lf,0

                mov     es, PSP                 ;Point ES at PSP
                lea     bx, CmdLn               ;Point at command line
PrintLoop:      print
                byte    cr,lf,'Item: ',0
                call    SkipDelimiters          ;Skip over leading delimiters
PrtLoop2:       mov     al, es:[bx]             ;Get next character
                call    TestDelimiter           ;Is it a delimiter?
                jz      EndOfToken              ;Quit this loop if it is
                putc                            ;Print char if not.
                inc     bx                      ;Move on to next character
                jmp     PrtLoop2

EndOfToken:     cmp     al, cr                  ;Carriage return?
                jne     PrintLoop               ;Repeat if not end of line

                print
                byte    cr,lf,lf
                byte    'End of command line',cr,lf,lf,0
                ExitPgm
MainPgm         endp

; The following subroutine sets the zero flag if the character in
; the AL register is one of DOS' six delimiter characters,
; otherwise the zero flag is returned clear. This allows us to use
; the JE/JNE instructions afterwards to test for a delimiter.

TestDelimiter   proc    near
                cmp     al, ' '
                jz      ItsOne
                cmp     al,','
                jz      ItsOne
                cmp     al,Tab
                jz      ItsOne
                cmp     al,';'
                jz      ItsOne
                cmp     al,'='
                jz      ItsOne
                cmp     al, cr
ItsOne:         ret
TestDelimiter   endp

; SkipDelimiters skips over leading delimiters on the command
; line. It does not, however, skip the carriage return at the end
; of a line since this character is used as the terminator in the
; main program.

SkipDelimiters  proc    near
                dec     bx                      ;To offset INC BX below
SDLoop:          inc    bx                      ;Move on to next character.
                mov     al, es:[bx]             ;Get next character
                cmp     al, 0dh                 ;Don't skip if CR.
                jz      QuitSD
                call    TestDelimiter           ;See if it's some other
                jz      SDLoop                  ; delimiter and repeat.
QuitSD:         ret
SkipDelimiters  endp

cseg            ends

dseg            segment byte public 'data'

PSP             word    ?                       ;Program segment prefix
dseg            ends
```

```
sseg            segment  byte stack 'stack'
stk             word     0ffh dup (?)
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end      MainPgm
```

Once you can scan the command line (that is, separate out the lexemes), the next step is to parse it. For most programs, parsing the command line is an extremely trivial process. If the program accepts only a single filename, all you've got to do is grab the first lexeme on the command line, slap a zero byte onto the end of it (perhaps moving it into your data segment), and use it as a filename. The following assembly language example modifies the hex dump routine presented earlier so that it gets its filename from the command line rather than hard-coding the filename into the program:

```
                include    stdlib.a
                includelib stdlib.lib
cseg            segment    byte public 'CODE'
                assume     cs:cseg, ds:dseg, es:dseg, ss:sseg

; Note CR and LF are already defined in STDLIB.A

tab             equ      09h

MainPgm         proc     far

; Properly set up the segment registers:

                mov      ax, seg dseg
                mov      es, ax                 ;Leave DS pointing at PSP

;-----------------------------------------------------------------
;
; First, parse the command line to get the filename:

                mov      si, 81h                ;Pointer to command line
                lea      di, FileName           ;Pointer to FileName buffer
SkipDelimiters:
                lodsb                           ;Get next character
                call     TestDelimiter
                je       SkipDelimiters

; Assume that what follows is an actual filename

                dec      si                     ;Point at 1st char of name
GetFName:       lodsb
                cmp      al, 0dh
                je       GotName
                call     TestDelimiter
                je       GotName
                stosb                           ;Save character in file name
                jmp      GetFName

; We're at the end of the filename, so zero-terminate it as
; required by DOS.

GotName:        mov      byte ptr es:[di], 0
                mov      ax, es                 ;Point DS at DSEG
                mov      ds, ax

; Now process the file

                mov      ah, 3dh
                mov      al, 0                  ;Open file for reading
                lea      dx, Filename           ;File to open
                int      21h
                jnc      GoodOpen
                print
                byte     'Cannot open file, aborting program...',cr,0
                jmp      PgmExit

GoodOpen:       mov      FileHandle, ax         ;Save file handle
```

```
                mov     Position, 0             ;Initialize file position
ReadFileLp:     mov     al, byte ptr Position
                and     al, 0Fh                 ;Compute (Position MOD 16)
                jnz     NotNewLn                ;Every 16 bytes start a line
                putcr
                mov     ax, Position            ;Print offset into file
                xchg    al, ah
                puth
                xchg    al, ah
                puth
                print
                byte    ': ',0

NotNewLn:       inc     Position                ;Increment character count
                mov     bx, FileHandle
                mov     cx, 1                   ;Read one byte
                lea     dx, buffer              ;Place to store that byte
                mov     ah, 3Fh                 ;Read operation
                int     21h
                jc      BadRead
                cmp     ax, 1                   ;Reached EOF?
                jnz     AtEOF
                mov     al, Buffer              ;Get the character read and
                puth                            ; print it in hex
                mov     al, ' '                 ;Print a space between values
                putc
                jmp     ReadFileLp

BadRead:        print
                byte    cr, lf
                byte    'Error reading data from file, aborting.'
                byte    cr,lf,0

AtEOF:          mov     bx, FileHandle          ;Close the file
                mov     ah, 3Eh
                int     21h

;-----------------------------------------------------------------

PgmExit:        ExitPgm
MainPgm         endp

TestDelimiter   proc    near
                cmp     al, ' '
                je      xit
                cmp     al, ','
                je      xit
                cmp     al, Tab
                je      xit
                cmp     al, ';'
                je      xit
                cmp     al, '='
xit:            ret
TestDelimiter   endp
cseg            ends

dseg            segment byte public 'data'

PSP             word    ?
Filename        byte    64 dup (0)              ;Filename to dump
FileHandle      word    ?
Buffer          byte    ?
Position        word    0

dseg            ends

sseg            segment byte stack 'stack'
stk             word    0ffh dup (?)
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
```

```
                end     MainPgm
```

The following example demonstrates several concepts dealing with command line parameters. This program copies one file to another. If the "/U" switch is supplied (somewhere) on the command line, all of the lower case characters in the file are converted to upper case before being written to the destination file. Another feature of this code is that it will prompt the user for any missing filenames, much like the MASM and LINK programs will prompt you for filename if you haven't supplied any.

```
                include     stdlib.a
                includelib  stdlib.lib

cseg            segment     byte public 'CODE'
                assume      cs:cseg, ds:nothing, es:dseg, ss:sseg

; Note: The constants CR (0dh) and LF (0ah) appear within the
; stdlib.a include file.

tab             equ     09h

MainPgm         proc    far

; Properly set up the segment registers:

                mov     ax, seg dseg
                mov     es, ax                  ;Leave DS pointing at PSP

;-----------------------------------------------------------------

; First, parse the command line to get the filename:

                mov     es:GotName1, 0          ;Init flags that tell us if
                mov     es:GotName2, 0          ; we've parsed the filenames
                mov     es:ConvertLC,0          ; and the "/U" switch.

; Okay, begin scanning and parsing the command line

                mov     si, 81h                 ;Pointer to command line
SkipDelimiters:
                lodsb                           ;Get next character
                call    TestDelimiter
                je      SkipDelimiters

; Determine if this is a filename or the /U switch

                cmp     al, '/'
                jnz     MustBeFN

; See if it's "/U" here-

                lodsb
                and     al, 5fh                 ;Convert "u" to "U"
                cmp     al, 'U'
                jnz     NotGoodSwitch
                lodsb                           ;Make sure next char is
                cmp     al, cr                  ; a delimiter of some sort
                jz      GoodSwitch
                call    TestDelimiter
                jne     NotGoodSwitch

; Okay, it's "/U" here.

GoodSwitch:     mov     es:ConvertLC, 1         ;Convert LC to UC
                dec     si                      ;Back up in case it's CR
                jmp     SkipDelimiters          ;Move on to next item.

; If a bad switch was found on the command line, print an error
; message and abort-

NotGoodSwitch:
                print
                byte    cr,lf
                byte    'Illegal switch, only "/U" is allowed!',cr,lf
                byte    'Aborting program execution.',cr,lf,0
                jmp     PgmExit

; If it's not a switch, assume that it's a valid filename and
; handle it down here-
```

```
MustBeFN:       cmp     al, cr                  ;See if at end of cmd line
                je      EndOfCmdLn

; See if it's filename one, two, or if too many filenames have been
; specified-

                cmp     es:GotName1, 0
                jz      Is1stName
                cmp     es:GotName2, 0
                jz      Is2ndName

; More than two filenames have been entered, print an error message
; and abort.

                print
                byte    cr,lf
                byte    'Too many filenames specified.',cr,lf
                byte    'Program aborting...',cr,lf,lf,0
                jmp     PgmExit

; Jump down here if this is the first filename to be processed-

Is1stName:      lea     di, FileName1
                mov     es:GotName1, 1
                jmp     ProcessName

Is2ndName:      lea     di, FileName2
                mov     es:GotName2, 1
ProcessName:
                stosb                           ;Store away character in name
                lodsb                           ;Get next char from cmd line
                cmp     al, cr
                je      NameIsDone
                call    TestDelimiter
                jne     ProcessName

NameIsDone:     mov     al, 0                   ;Zero terminate filename
                stosb
                dec     si                      ;Point back at previous char
                jmp     SkipDelimiters          ;Try again.

; When the end of the command line is reached, come down here and
; see if both filenames were specified.

                assume     ds:dseg

EndOfCmdLn:     mov     ax, es                  ;Point DS at DSEG
                mov     ds, ax

; We're at the end of the filename, so zero-terminate it as
;  required by DOS.

GotName:        mov     ax, es                  ;Point DS at DSEG
                mov     ds, ax

; See if the names were supplied on the command line.
; If not, prompt the user and read them from the keyboard

                cmp     GotName1, 0             ;Was filename #1 supplied?
                jnz     HasName1
                mov     al, '1'                 ;Filename #1
                lea     si, Filename1
                call    GetName                 ;Get filename #1

HasName1:       cmp     GotName2, 0             ;Was filename #2 supplied?
                jnz     HasName2
                mov     al, '2'                 ;If not, read it from kbd.
                lea     si, FileName2
                call    GetName

; Okay, we've got the filenames, now open the files and copy the
; source file to the destination file.

HasName2        mov     ah, 3dh
                mov     al, 0                   ;Open file for reading
                lea     dx, Filename1           ;File to open
```

```
                int     21h
                jnc     GoodOpen1

                print
                byte    'Cannot open file, aborting program...',cr,lf,0
                jmp     PgmExit

; If the source file was opened successfully, save the file handle.

GoodOpen1:      mov     FileHandle1, ax         ;Save file handle

; Open (CREATE, actually) the second file here.

                mov     ah, 3ch                 ;Create file
                mov     cx, 0                   ;Standard attributes
                lea     dx, Filename2           ;File to open
                int     21h
                jnc     GoodCreate

; Note: the following error code relies on the fact that DOS
; automatically closes any open source files when the program
; terminates.

                print
                byte    cr,lf
                byte    'Cannot create new file, aborting operation'
                byte    cr,lf,lf,0
                jmp     PgmExit

GoodCreate:     mov     FileHandle2, ax         ;Save file handle

; Now process the files

CopyLoop:       mov     ah, 3Fh                 ;DOS read opcode
                mov     bx, FileHandle1         ;Read from file #1
                mov     cx, 512                 ;Read 512 bytes
                lea     dx, buffer              ;Buffer for storage
                int     21h
                jc      BadRead
                mov     bp, ax                  ;Save # of bytes read

                cmp     ConvertLC,0             ;Conversion option active?
                jz      NoConversion

; Convert all LC in buffer to UC-

                mov     cx, 512
                lea     si, Buffer
                mov     di, si
ConvertLC2UC:
                lodsb
                cmp     al, 'a'
                jb      NoConv
                cmp     al, 'z'
                ja      NoConv
                and     al, 5fh
NoConv:         stosb
                loop    ConvertLC2UC

NoConversion:
                mov     ah, 40h                 ;DOS write opcode
                mov     bx, FileHandle2         ;Write to file #2
                mov     cx, bp                  ;Write however many bytes
                lea     dx, buffer              ;Buffer for storage
                int     21h
                jc      BadWrite
                cmp     ax, bp                  ;Did we write all of the
                jnz     jDiskFull               ; bytes?
                cmp     bp, 512                 ;Were there 512 bytes read?
                jz      CopyLoop
                jmp     AtEOF
jDiskFull:      jmp     DiskFull

; Various error messages:

BadRead:        print
```

```
                byte    cr,lf
                byte    'Error while reading source file, aborting '
                byte    'operation.',cr,lf,0
                jmp     AtEOF

BadWrite:       print
                byte    cr,lf
                byte    'Error while writing destination file, aborting'
                byte    ' operation.',cr,lf,0
                jmp     AtEOF

DiskFull:       print
                byte    cr,lf
                byte    'Error, disk full.  Aborting operation.',cr,lf,0

AtEOF:          mov     bx, FileHandle1         ;Close the first file
                mov     ah, 3Eh
                int     21h
                mov     bx, FileHandle2         ;Close the second file
                mov     ah, 3Eh
                int     21h

PgmExit:        ExitPgm
MainPgm         endp

TestDelimiter   proc    near
                cmp     al, ' '
                je      xit
                cmp     al, ','
                je      xit
                cmp     al, Tab
                je      xit
                cmp     al, ';'
                je      xit
                cmp     al, '='
xit:            ret
TestDelimiter   endp

; GetName- Reads a filename from the keyboard.  On entry, AL
; contains the filename number and DI points at the buffer in ES
; where the zero-terminated filename must be stored.

GetName         proc    near
                print
                byte    'Enter filename #',0
                putc
                mov     al, ':'
                putc
                gets
                ret
GetName         endp
cseg            ends

dseg            segment byte public 'data'

PSP             word    ?
Filename1       byte    128 dup (?);Source filename
Filename2       byte    128 dup (?);Destination filename
FileHandle1     word    ?
FileHandle2     word    ?
GotName1        byte    ?
GotName2        byte    ?
ConvertLC       byte    ?
Buffer          byte    512 dup (?)

dseg            ends

sseg            segment   byte stack 'stack'
stk             word    0ffh dup (?)
sseg            ends

zzzzzzseg       segment   para public 'zzzzzz'
LastBytes       byte      16 dup (?)
zzzzzzseg       ends
                end     MainPgm
```

As you can see, there is more effort expended processing the command line parameters than actually copying the files!

13.3.13 ARGC and ARGV

The UCR Standard Library provides two routines, argc and argv, which provide easy access to command line parameters. Argc (*argument count*) returns the number of items on the command line. Argv (*argument vector*) returns a pointer to a specific item in the command line.

These routines break up the command line into lexemes using the standard delimiters. As per MS-DOS convention, argc and argv treat any string surrounded by quotation marks on the command line as a single command line item.

Argc will return in cx the number of command line items. Since MS-DOS does not include the program name on the command line, this count does not include the program name either. Furthermore, redirection operands (“>filename” and “<filename”) and items to the right of a pipe (“| command”) do not appear on the command line either. As such, argc does not count these, either.

Argv returns a pointer to a string (allocated on the heap) of a specified command line item. To use argv you simply load ax with a value between one and the number returned by argc and execute the argv routine. On return, es:di points at a string containing the specified command line option. If the number in ax is greater than the number of command line arguments, then argv returns a pointer to an empty string (i.e., a zero byte). Since argv calls malloc to allocate storage on the heap, there is the possibility that a memory allocation error will occur. Argv returns the carry set if a memory allocation error occurs. Remember to free the storage allocated to a command line parameter after you are through with it.

Example: The following code echoes the command line parameters to the screen.

```
                include     stdlib.a
                includelib  stdlib.lib

dseg            segment   para public 'data'

ArgCnt          word      0

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov       ax, dseg
                mov       ds, ax
                mov       es, ax

; Must call the memory manager initialization routine if you use
; any routine which calls malloc!  ARGV is a good example of a
; routine which calls malloc.

                meminit

                argc                ;Get the command line arg count.
                jcxz      Quit      ;Quit if no cmd ln args.
                mov       ArgCnt, 1 ;Init Cmd Ln count.
PrintCmds:      printf              ;Print the item.
                byte      "\n%2d: ",0
                dword     ArgCnt

                mov       ax, ArgCnt ;Get the next command line guy.
                argv
                puts
                inc       ArgCnt    ;Move on to next arg.
                loop      PrintCmds ;Repeat for each arg.
                putcr

Quit:           ExitPgm             ;DOS macro to quit program.
```

```
Main            endp
cseg            ends

sseg            segment    para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

;zzzzzzseg is required by the standard library routines.

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        Main
```

13.4 UCR Standard Library File I/O Routines

Although MS-DOS' file I/O facilities are not too bad, the UCR Standard Library provides a file I/O package which makes blocked sequential I/O as easy as character at a time file I/O. Furthermore, with a tiny amount of effort, you can use all the StdLib routines like printf, print, puti, puth, putc, getc, gets, etc., when performing file I/O. This greatly simplifies text file operations in assembly language.

Note that record oriented, or binary I/O, is probably best left to pure DOS. any time you want to do random access within a file. The Standard Library routines really only support sequential text I/O. Nevertheless, this is the most common form of file I/O around, so the Standard Library routines are quite useful indeed.

The UCR Standard Library provides eight file I/O routines: fopen, fcreate, fclose, fgetc, fread, fputc, and fwrite. Fgetc and fputc perform character at a time I/O, fread and fwrite let you read and write blocks of data, the other four functions perform the obvious DOS operations.

The UCR Standard Library uses a special *file variable* to keep track of file operations. There is a special record type, *FileVar*, declared in stdlib.a[8]. When using the StdLib file I/O routines you must create a variable of type FileVar for every file you need open at the same time. This is very easy, just use a definition of the form:

```
MyFileVar   FileVar             {}
```

Please note that a Standard Library file variable *is not* the same thing as a DOS file handle. It is a structure which contains the DOS file handle, a buffer (for blocked I/O), and various index and status variables. The internal structure of this type is of no interest (remember data encapsulation!) except to the implementor of the file routines. You will pass the address of this file variable to the various Standard Library file I/O routines.

13.4.1 Fopen

Entry parameters:	ax-	File open mode 0- File opened for reading 1- File opened for writing
	dx:si-	Points at a zero terminated string containing the filename.
	es:di-	Points at a StdLib file variable.

Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fopen opens a sequential text file for reading *or* writing. Unlike DOS, you cannot open a file for reading and writing. Furthermore, this is a sequential text file which does not support random access. Note that the file must exist or fopen will return an error. This is even true when you open the file for writing.

8. Actually, it's declared in *file.a*. Stdlib.a includes file.a so this definition appears inside stdlib.a as well.

Note that if you open a file for writing and that file already exists, any data written to the file will overwrite the existing data. When you close the file, any data appearing in the file after the data you wrote will still be there. If you want to erase the existing file before writing data to it, use the fcreate function.

13.4.2 Fcreate

Entry parameters: dx:si- Points at a zero terminated string containing the filename.
es:di- Points at a StdLib file variable.

Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fcreate creates a new file and opens it for writing. If the file already exists, fcreate deletes the existing file and creates a new one. It initializes the file variable for output but is otherwise identical to the fopen call.

13.4.3 Fclose

Entry parameters: es:di- Points at a StdLib file variable.
Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fclose closes a file and updates any internal housekeeping information. *It is very important that you close all files opened with fopen or fcreate using this call.* When making DOS file calls, if you forget to close a file DOS will automatically do that for you when your program terminates. However, the StdLib routines cache up data in internal buffers. the fclose call automatically flushes these buffers to disk. If you exit your program without calling fclose, you may lose some data written to the file but not yet transferred from the internal buffer to the disk.

If you are in an environment where it is possible for someone to abort the program without giving you a chance to close the file, you should call the fflush routines (see the next section) on a regular basis to avoid losing too much data.

13.4.4 Fflush

Entry parameters: es:di- Points at a StdLib file variable.
Exit parameters: If the carry is set, **ax** contains the returned DOS error code (see DOS open function).

This routine immediately writes any data in the internal file buffer to disk. Note that you should only use this routine in conjunction with files opened for writing (or opened by fcreate). If you write data to a file and then need to leave the file open, but inactive, for some time period, you should perform a flush operation in case the program terminates abnormally.

13.4.5 Fgetc

Entry parameters: es:di- Points at a StdLib file variable.
Exit parameters: If the carry flag is clear, al contains the character read from the file.
If the carry is set, ax contains the returned DOS error code (see DOS open function).
ax will contain zero if you attempt to read beyond the end of file.

Fgetc reads a single character from the file and returns this character in the al register. Unlike calls to DOS, single character I/O using fgetc is relatively fast since the StdLib routines use blocked I/O. Of course, multiple calls to fgetc will never be faster than a call to fread (see the next section), but the performance is not too bad.

Fgetc is very flexible. As you will see in a little bit, you may redirect the StdLib input routines to read their data from a file using fgetc. This lets you use the higher level routines like gets and getsm when reading data from a file.

13.4.6 Fread

Entry parameters: es:di- Points at a StdLib file variable.
dx:si- Points at an input data buffer.
cx- Contains a byte count.
Exit parameters: If the carry flag is clear, ax contains the actual number of bytes read from the file.
If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fread is very similar to the DOS read command. It lets you read a block of bytes, rather than just one byte, from a file. Note that if all you are doing is reading a block of bytes from a file, the DOS call is slightly more efficient than fread. However, if you have a mixture of single byte reads and multi-byte reads, the combination of fread and fgetc work very well.

As with the DOS read operation, if the byte count returned in ax does not match the value passed in the cx register, then you've read the remaining bytes in the file. When this occurs, the next call to fread or fgetc will return an EOF error (carry will be set and ax will contain zero). Note that fread does not return EOF unless there were zero bytes read from the file.

13.4.7 Fputc

Entry parameters: es:di- Points at a StdLib file variable.
al- Contains the character to write to the file.

Exit parameters: If the carry is set, ax contains the returned DOS error code (see DOS open function).

Fputc writes a single character (in al) to the file specified by the file variable whose address is in es:di. This call simply adds the character in al to an internal buffer (part of the file variable) until the buffer is full. Whenever the buffer is filled or you call fflush (or close the file with fclose), the file I/O routines write the data to disk.

13.4.8 Fwrite

Entry parameters: es:di- Points at a StdLib file variable.
dx:si- Points at an output data buffer.
cx- Contains a byte count.
Exit parameters: If the carry flag is clear, ax contains the actual number of bytes written to the file.
If the carry is set, ax contains the returned DOS error code (see DOS open function).

Like fread, fwrite works on blocks of bytes. It lets you write a block of bytes to a file opened for writing with fopen or fcreate.

13.4.9 Redirecting I/O Through the StdLib File I/O Routines

The Standard Library provides very few file I/O routines. Fputc and fwrite are the only two output routines, for example. The "C" programming language standard library (on which the UCR Standard Library is based) provides many routines like *fprintf, fputs, fscanf,* etc. None of these are necessary in the UCR Standard Library because the UCR library provides an I/O redirection mechanism that lets you reuse all existing I/O routines to perform file I/O.

The UCR Standard Library putc routine consists of a single jmp instruction. This instruction transfers control to some actual output routine via an indirect address internal to the putc code. Normally, this pointer variable points at a piece of code which writes the character in the al register to the DOS standard output device. However, the Standard Library also provides four routines which let you manipulate this indirect pointer. By changing this pointer you can redirect the output from its current routine to a routine of your choosing. *All* Standard Library output routines (e.g., printf, puti, puth, puts) call putc to output individual characters. Therefore, redirecting the putc routine affects all the output routines.

Likewise, the getc routine is nothing more than an indirect jmp whose pointer variable normally points at a piece of code which reads data from the DOS standard input. Since all Standard Library input routines call the getc function to read each character you can redirect file input in a manner identical to file output.

The Standard Library *GetOutAdrs, SetOutAdrs, PushOutAdrs,* and *PopOutAdrs* are the four main routines which manipulate the output redirection pointer. GetOutAdrs returns the address of the current output routine in the es:di registers. Conversely, SetOutAdrs expects you to pass the address of a new output routine in the es:di registers and it stores this address into the output pointer. PushOutAdrs and PopOutAdrs push and pop the pointer on an internal stack. These do not use the 80x86's hardware stack. You are limited to a small number of pushes and pops. Generally, you shouldn't count on being able to push more than four of these addresses onto the internal stack without overflowing it.

GetInAdrs, SetInAdrs, PushInAdrs, and *PopInAdrs* are the complementary routines for the input vector. They let you manipulate the input routine pointer. Note that the stack for PushInAdrs/PopInAdrs is not the same as the stack for PushOutAdrs/PopOutAdrs. Pushes and pops to these two stacks are independent of one another.

Normally, the output pointer (which we will henceforth refer to as the *output hook*) points at the Standard Library routine *PutcStdOut*[9]. Therefore, you can return the output hook to its normal initialization state at any time by executing the statements[10]:

```
mov     di, seg SL_PutcStdOut
mov     es, di
mov     di, offset SL_PutcStdOut
SetOutAdrs
```

The PutcStdOut routine writes the character in the al register to the DOS standard output, which itself might be redirected to some file or device (using the ">" DOS redirection operator). If you want to make sure your output is going to the video display, you can always call the PutcBIOS routine which calls the BIOS directly to output a character[11]. You can force all Standard Library output to the *standard error device* using a code sequence like:

```
mov     di, seg SL_PutcBIOS
mov     es, di
mov     di, offset SL_PutcBIOS
SetOutAdrs
```

Generally, you would not simply blast the output hook by storing a pointer to your routine over the top of whatever pointer was there and then restoring the hook to PutcStdOut upon completion. Who knows if the hook was pointing at PutcStdOut in the first place? The best solution is to use the Standard Library PushOutAdrs and PopOutAdrs routines to preserve and restore the previous hook. The following code demonstrates a *gentler* way of modifying the output hook:

---

9. Actually, the routine is *SL_PutcStdOut*. The Standard Library macro by which you would normally call this routine is PutcStdOut.

10. If you do not have any calls to PutcStdOut in your program, you will also need to add the statement "externdef SL_PutcStdOut:far" to your program.

11. It is possible to redirect even the BIOS output, but this is rarely done and not easy to do from DOS.

```
                PushOutAdrs             ;Save current output routine.
                mov     di, seg Output_Routine
                mov     es, di
                mov     di, offset Output_Routine
                SetOutAdrs

        <Do all output to Output_Routine here>

                PopOutAdrs              ;Restore previous output routine.
```

Handle input in a similar fashion using the corresponding input hook access routines and the SL_GetcStdOut and SL_GetcBIOS routines. Always keep in mind that there are a limited number of entries on the input and output hook stacks so what how many items you push onto these stacks without popping anything off.

To redirect output to a file (or redirect input from a file) you must first write a short routine which writes (reads) a single character from (to) a file. This is very easy. The code for a subroutine to output data to a file described by file variable *OutputFile* is

```
ToOutput        proc    far
                push    es
                push    di

; Load ES:DI with the address of the OutputFile variable. This
; code assumes OutputFile is of type FileVar, not a pointer to
; a variable of type FileVar.

                mov     di, seg OutputFile
                mov     es, di
                mov     di, offset OutputFile

; Output the character in AL to the file described by "OutputFile"

                fputc

                pop     di
                pop     es
                ret
ToOutput        endp
```

Now with only one additional piece of code, you can begin writing data to an output file using all the Standard Library output routines. That is a short piece of code which redirects the output hook to the "ToOutput" routine above:

```
SetOutFile      proc
                push    es
                push    di

                PushOutAdrs                     ;Save current output hook.
                mov     di, seg ToOutput
                mov     es, di
                mov     di, offset ToOutput
                SetOutAdrs

                pop     di
                pop     es
                ret
SetOutFile      endp
```

There is no need for a separate routine to restore the output hook to its previous value; PopOutAdrs will handle that task by itself.

13.4.10 A File I/O Example

The following piece of code puts everything together from the last several sections. This is a short program which adds line numbers to a text file. This program expects two command line parameters: an input file and an output file. It copies the input file to the output file while appending line numbers to the beginning of each line in the output file. This code demonstrates the use of argc, argv, the Standard Library file I/O routines, and I/O redirection.

```
; This program copies the input file to the output file and adds
; line numbers while it is copying the file.

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment     para public 'data'

ArgCnt          word        0
LineNumber      word        0
DOSErrorCode    word        0
InFile          dword       ?                   ;Ptr to Input file name.
OutFile         dword       ?                   ;Ptr to Output file name
InputLine       byte        1024 dup (0)        ;Input/Output data buffer.
OutputFile      FileVar     {}
InputFile       FileVar     {}

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

; ReadLn- Reads a line of text from the input file and stores the
;         data into the InputLine buffer:

ReadLn          proc
                push    ds
                push    es
                push    di
                push    si
                push    ax

                mov     si, dseg
                mov     ds, si
                mov     si, offset InputLine
                lesi    InputFile

GetLnLp:
                fgetc
                jc      RdLnDone                ;If some bizzarre error.
                cmp     ah, 0                   ;Check for EOF.
                je      RdLnDone                ;Note:carry is set.
                mov     ds:[si], al
                inc     si
                cmp     al, lf                  ;At EOLN?
                jne     GetLnLp
                dec     si                      ;Back up before LF.
                cmp     byte ptr ds:[si-1], cr ;CR before LF?
                jne     RdLnDone
                dec     si                      ;If so, skip it too.

RdLnDone:       mov     byte ptr ds:[si], 0 ;Zero terminate.
                pop     ax
                pop     si
                pop     di
                pop     es
                pop     ds
                ret
ReadLn          endp

; MyOutput- Writes the single character in AL to the output file.

MyOutput        proc    far
                push    es
                push    di
                lesi    OutputFile
                fputc
                pop     di
                pop     es
                ret
MyOutput        endp

; The main program which does all the work:

Main            proc
```

```
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

; Must call the memory manager initialization routine if you use
; any routine which calls malloc!  ARGV is a good example of a
; routine calls malloc.

                meminit

; We expect this program to be called as follows:
;               fileio file1, file2
; anything else is an error.

                argc
                cmp     cx, 2       ;Must have two parameters.
                je      Got2Parms
BadParms:       print
                byte    "Usage: FILEIO infile, outfile",cr,lf,0
                jmp     Quit

; Okay, we've got two parameters, hopefully they're valid names.
; Get copies of the filenames and store away the pointers to them.

Got2Parms:      mov     ax, 1       ;Get the input filename
                argv
                mov     word ptr InFile, di
                mov     word ptr InFile+2, es

                mov     ax, 2       ;Get the output filename
                argv
                mov     word ptr OutFile, di
                mov     word ptr OutFile+2, es

; Output the filenames to the standard output device

                printf
                byte    "Input file: %^s\n"
                byte    "Output file: %^s\n",0
                dword   InFile, OutFile

; Open the input file:

                lesi    InputFile
                mov     dx, word ptr InFile+2
                mov     si, word ptr InFile
                mov     ax, 0
                fopen
                jnc     GoodOpen
                mov     DOSErrorCode, ax
                printf
                byte    "Could not open input file, DOS: %d\n",0
                dword   DOSErrorCode
                jmp     Quit

; Create a new file for output:

GoodOpen:       lesi    OutputFile
                mov     dx, word ptr OutFile+2
                mov     si, word ptr OutFile
                fcreate
                jnc     GoodCreate
                mov     DOSErrorCode, AX
                printf
                byte    "Could not open output file, DOS: %d\n",0
                dword   DOSErrorCode
                jmp     Quit

; Okay, save the output hook and redirect the output.

GoodCreate:     PushOutAdrs
                lesi    MyOutput
                SetOutAdrs

WhlNotEOF:      inc     LineNumber

; Okay, read the input line from the user:
```

```
                call    ReadLn
                jc      BadInput

; Okay, redirect the output to our output file and write the last
; line read prefixed with a line number:

                printf
                byte    "%4d:   %s\n",0
                dword   LineNumber, InputLine
                jmp     WhlNotEOF

BadInput:       push    ax          ;Save error code.
                PopOutAdrs          ;Restore output hook.
                pop     ax          ;Retrieve error code.
                test    ax, ax      ;EOF error? (AX = 0)
                jz      CloseFiles
                mov     DOSErrorCode, ax
                printf
                byte    "Input error, DOS: %d\n",0
                dword   LineNumber

; Okay, close the files and quit:

CloseFiles:     lesi    OutputFile
                fclose
                lesi    InputFile
                fclose

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       byte       16 dup (?)
zzzzzzseg       ends
                end        Main
```

13.5 Sample Program

If you want to use the Standard Library's output routines (putc, print, printf, etc.) to output data to a file, you can do so by manually redirecting the output before and after each call to these routines. Unfortunately, this can be a lot of work if you mix interactive I/O with file I/O. The following program presents several macros that simplify this task for you.

```
; FileMacs.asm
;
; This program presents a set of macros that make file I/O with the
; Standard Library even easier to do.
;
; The main program writes a multiplication table to the file "MyFile.txt".

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment  para public 'data'

CurOutput       dword    ?

Filename        byte     "MyFile.txt",0

i               word     ?
j               word     ?
```

```
TheFile         filevar  {}

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


; For-Next macros from Chapter Eight.
; See Chapter Eight for details on how this works.

ForLp           macro    LCV, Start, Stop
                local    ForLoop

                ifndef   $$For&LCV&
$$For&LCV&=     0
                else
$$For&LCV&=     $$For&LCV& + 1
                endif

                mov      ax, Start
                mov      LCV, ax

ForLoop         textequ  @catstr($$For&LCV&, %$$For&LCV&)
&ForLoop&:
                mov      ax, LCV
                cmp      ax, Stop
                jg       @catstr($$Next&LCV&, %$$For&LCV&)
                endm



Next            macro    LCV
                local    NextLbl
                inc      LCV
                jmp      @catstr($$For&LCV&, %$$For&LCV&)
NextLbl         textequ  @catstr($$Next&LCV&, %$$For&LCV&)
&NextLbl&:
                endm


; File I/O macros:
;
;
; SetPtr sets up the CurOutput pointer variable.  This macro is called
; by the other macros, it's not something you would normally call directly.
; Its whole purpose in life is to shorten the other macros and save a little
; typing.

SetPtr          macro    fvar
                push     es
                push     di

                mov      di, offset fvar
                mov      word ptr CurOutput, di
                mov      di, seg fvar
                mov      word ptr CurOutput+2, di

                PushOutAdrs
                lesi     FileOutput
                SetOutAdrs
                pop      di
                pop      es
                endm
;
;
;
; fprint-       Prints a string to the display.
```

```
;
; Usage:
;                fprint   filevar,"String or bytes to print"
;
; Note: you can supply optional byte or string data after the string above by
;      enclosing the data in angle brackets, e.g.,
;
;                fprint   filevar,<"string to print",cr,lf>
;
; Do *NOT* put a zero terminating byte at the end of the string, the fprint
; macro will do that for you automatically.

fprint           macro    fvar:req, string:req
                 SetPtr   fvar

                 print
                 byte     string
                 byte     0

                 PopOutAdrs
                 endm

; fprintf-       Prints a formatted string to the display.
; fprintff-      Like fprintf, but handles floats as well as other items.
;
; Usage:
;                fprintf  filevar,"format string", optional data values
;                fprintff filevar,"format string", optional data values
; Examples:
;
;      fprintf   FileVariable,"i=%d, j=%d\n", i, j
;      fprintff  FileVariable,"f=%8.2f, i=%d\n", f, i
;
; Note: if you want to specify a list of strings and bytes for the format
;       string, just surround the items with an angle bracket, e.g.,
;
;      fprintf FileVariable, <"i=%d, j=%d",cr,lf>, i, j
;
;

fprintf          macro    fvar:req, FmtStr:req, Operands:vararg
                 setptr   fvar

                 printf
                 byte     FmtStr
                 byte     0

                 for      ThisVal, <Operands>
                 dword    ThisVal
                 endm

                 PopOutAdrs
                 endm

fprintff         macro    fvar:req, FmtStr:req, Operands:vararg
                 setptr   fvar

                 printff
                 byte     FmtStr
                 byte     0

                 for      ThisVal, <Operands>
                 dword    ThisVal
                 endm

                 PopOutAdrs
                 endm


; F-   This is a generic macro that converts stand-alone (no code stream parms)
```

```
;       stdlib functions into file output routines.  Use it with putc, puts,
;       puti, putu, putl, putisize, putusize, putlsize, putcr, etc.
;
; Usage:
;
;       F         StdLibFunction, FileVariable
;
; Examples:
;
;       mov       al, 'A'
;       F         putc, TheFile
;       mov       ax, I
;       mov       cx, 4
;       F         putisize, TheFile


F               macro     func:req, fvar:req
                setptr    fvar
                func
                PopOutAdrs
                endm

; WriteLn- Quick macro to handle the putcr operation (since this code calls
; putcr so often).

WriteLn         macro     fvar:req
                F         putcr, fvar
                endm


; FileOutput- Writes the single character in AL to an output file.
; The macros above redirect the standard output to this routine
; to print data to a file.

FileOutput      proc      far
                push      es
                push      di
                push      ds
                mov       di, dseg
                mov       ds, di

                les       di, CurOutput
                fputc

                pop       ds
                pop       di
                pop       es
                ret
FileOutput      endp


; A simple main program that tests the code above.
; This program writes a multiplication table to the file "MyFile.txt"

Main            proc
                mov       ax, dseg
                mov       ds, ax
                mov       es, ax
                meminit

; Rewrite(TheFile, FileName);

                ldxi      FileName
                lesi      TheFile
                fcreate

; writeln(TheFile);
; writeln(TheFile,'    ');
; for i := 0 to 5 do write(TheFile,'|',i:4,' ');
; writeln(TheFile);
```

```
                WriteLn  TheFile
                fprint   TheFile,"    "

                forlp    i,0,5
                fprintf  TheFile, "|%4d ", i
                next     i
                WriteLn  TheFile

; for j := -5 to 5 do begin
;
;      write(TheFile,'----');
;      for i := 0 to 5 do write(TheFile, '+-----');
;      writeln(TheFile);
;
;      write(j:3, ' |');
;      for i := 0 to 5 do write(i*j:4, ' |);
;      writeln(TheFile);
;
; end;

                forlp    j,-5,5

                fprint   TheFile,"----"
                forlp    i,0,5
                fprintf  TheFile,"+-----"
                next     i
                fprint   TheFile,<"+",cr,lf>

                fprintf  TheFile, "%3d |", j

                forlp    i,0,5

                mov      ax, i
                imul     j
                mov      cx, 4
                F        putisize, TheFile
                fprint   TheFile, " |"

                next     i
                Writeln  TheFile

                next     j
                WriteLn  TheFile

; Close(TheFile);

                lesi     TheFile
                fclose


Quit:           ExitPgm              ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

13.6 Laboratory Exercises

The following three programs all do the same thing: they copy the file ”ex13_1.in” to the file “ex13_1.out”. The difference is the way they copy the files. The first program, ex13_1a, copies the data from the input file to the output file using character at a time I/O under DOS. The second program, ex13_1b, uses blocked I/O under DOS. The third program, ex13_1c, uses the Standard Library’s file I/O routines to copy the data.

Run these three programs and measure the amount of time they take to run[12]. **For your lab report:** report the running times and comment on the relative efficiencies of these data transfer methods. Is the loss of performance of the Standard Library routines (compared to block I/O) justified in terms of the ease of use of these routines? Explain.

```
; EX13_1a.asm
;
; This program copies one file to another using character at a time I/O.
; It is easy to write, read, and understand, but character at a time I/O
; is quite slow.  Run this program and time its execution.  Then run the
; corresponding blocked I/O exercise and compare the execution times of
; the two programs.

                include     stdlib.a
                includelib  stdlib.lib


dseg            segment  para public 'data'

FHndl           word     ?
FHndl2          word     ?
Buffer          byte     ?

FName           equ      this word
FNamePtr        dword    FileName

Filename        byte     "Ex13_1.in",0
Filename2       byte     "Ex13_1.out",0

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                mov      ah, 3dh       ;Open the input file
                mov      al, 0         ; for reading
                lea      dx, Filename  ;DS points at filename’s
                int      21h           ; segment
                jc       BadOpen
                mov      FHndl, ax     ;Save file handle


                mov      FName, offset Filename2 ;Set this up in case there
                mov      FName+2, seg FileName2  ; is an error during open.

                mov      ah, 3ch       ;Open the output file for writing
                mov      cx, 0         ; with normal file attributes
```

12. If you have a really fast machine you may want to make the ex13_1.in file larger (by copying and pasting data in the file) to make it larger.

```
                lea     dx, Filename2 ;Presume DS points at filename
                int     21h           ; segment
                jc      BadOpen
                mov     FHndl2, ax    ;Save file handle

LP:             mov     ah,3fh        ;Read data from the file
                lea     dx, Buffer    ;Address of data buffer
                mov     cx, 1         ;Read one byte
                mov     bx, FHndl     ;Get file handle value
                int     21h
                jc      ReadError
                cmp     ax, cx        ;EOF reached?
                jne     EOF

                mov     ah,40h        ;Write data to the file
                lea     dx, Buffer    ;Address of data buffer
                mov     cx, 1         ;Write one byte
                mov     bx, FHndl2    ;Get file handle value
                int     21h
                jc      WriteError
                jmp     LP            ;Read next byte

EOF:            mov     bx, FHndl
                mov     ah, 3eh       ;Close file
                int     21h
                jmp     Quit

ReadError:      printf
                byte    "Error while reading data from file '%s'.",cr,lf,0
                dword   FileName
                jmp     Quit

WriteError:     printf
                byte    "Error while writing data to file '%s'.",cr,lf,0
                dword   FileName2
                jmp     Quit

BadOpen:        printf
                byte    "Could not open '%^s'.  Make sure this file is “
                byte    “in the ",cr,lf
                byte    "current directory before attempting to run “
                byte    this program again.", cr,lf,0
                dword   FName

Quit:           ExitPgm               ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main


-------------------------------------------------


; EX13_1b.asm
;
; This program copies one file to another using blocked I/O.
; Run this program and time its execution.  Compare the execution time of
; this program against that of the character at a time I/O and the
; Standard Library File I/O example (ex13_1a and ex13_1c).

                include    stdlib.a
```

```
                includelib  stdlib.lib


dseg            segment  para public 'data'

; File handles for the files we will open.

FHndl           word     ?                    ;Input file handle
FHndl2          word     ?                    ;Output file handle

Buffer          byte     256 dup (?)          ;File buffer area

FName           equ      this word            ;Ptr to current file name
FNamePtr        dword    FileName

Filename        byte     "Ex13_1.in",0        ;Input file name
Filename2       byte     "Ex13_1.out",0       ;Output file name

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                mov      ah, 3dh              ;Open the input file
                mov      al, 0                ; for reading
                lea      dx, Filename         ;Presume DS points at
                int      21h                  ; filename's segment
                jc       BadOpen
                mov      FHndl, ax            ;Save file handle


                mov      FName, offset Filename2 ;Set this up in case there
                mov      FName+2, seg FileName2  ; is an error during open.

                mov      ah, 3ch       ;Open the output file for writing
                mov      cx, 0         ; with normal file attributes
                lea      dx, Filename2 ;Presume DS points at filename
                int      21h           ; segment
                jc       BadOpen
                mov      FHndl2, ax    ;Save file handle


; The following loop reads 256 bytes at a time from the file and then
; writes those 256 bytes to the output file.

LP:             mov      ah,3fh       ;Read data from the file
                lea      dx, Buffer   ;Address of data buffer
                mov      cx, 256      ;Read 256 bytes
                mov      bx, FHndl    ;Get file handle value
                int      21h
                jc       ReadError
                cmp      ax, cx       ;EOF reached?
                jne      EOF

                mov      ah, 40h      ;Write data to file
                lea      dx, Buffer   ;Address of output buffer
                mov      cx, 256      ;Write 256 bytes
                mov      bx, FHndl2   ;Output handle
                int      21h
                jc       WriteError
                jmp      LP           ;Read next block

; Note, just because the number of bytes read does not equal 256,
```

```
; don't get the idea we're through, there could be up to 255 bytes
; in the buffer still waiting to be processed.

EOF:            mov     cx, ax      ;Put # of bytes to write in CX.
                jcxz    EOF2        ;If CX is zero, we're really done.
                mov     ah, 40h     ;Write data to file
                lea     dx, Buffer  ;Address of output buffer
                mov     bx, FHndl2  ;Output handle
                int     21h
                jc      WriteError


EOF2:           mov     bx, FHndl
                mov     ah, 3eh     ;Close file
                int     21h
                jmp     Quit

ReadError:      printf
                byte    "Error while reading data from file '%s'.",cr,lf,0
                dword   FileName
                jmp     Quit

WriteError:     printf
                byte    "Error while writing data to file '%s'.",cr,lf,0
                dword   FileName2
                jmp     Quit

BadOpen:        printf
                byte    "Could not open '%^s'.  Make sure this file is in “
                byte    “the ",cr,lf
                byte    "current directory before attempting to run “
                byte    “this program again.", cr,lf,0
                dword   FName

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg           ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main

-------------------------------------------------

; EX13_1c.asm
;
; This program copies one file to another using the standard library
; file I/O routines.  The Standard Library file I/O routines let you do
; character at a time I/O, but they block up the data to transfer to improve
; system performance.  You should find that the execution time of this
; code is somewhere between blocked I/O (ex13_1b) and character at a time
; I/O (EX13_1a);  it will, however, be much closer to the block I/O time
; (probably about twice as long as block I/O).

                include    stdlib.a
                includelib stdlib.lib


dseg            segment  para public 'data'

InFile          filevar  {}
OutFile         filevar  {}

Filename        byte     "Ex13_1.in",0;Input file name
```

```
Filename2       byte    "Ex13_1.out",0;Output file name

dseg            ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; Open the input file:

                mov     ax, 0                   ;Open for reading
                ldxi    Filename
                lesi    InFile
                fopen
                jc      BadOpen

; Open the output file:

                mov     ax, 1                   ;Open for output
                ldxi    Filename2
                lesi    OutFile
                fcreate
                jc      BadCreate

; Copy the input file to the output file:

CopyLp:         lesi    InFile
                fgetc
                jc      GetDone

                lesi    OutFile
                fputc
                jmp     CopyLp

BadOpen:        printf
                byte    "Error opening '%s'",cr,lf,0
                dword   Filename
                jmp     Quit

BadCreate:      printf
                byte    "Error creating '%s'",cr,lf,0
                dword   Filename2
                jmp     CloseIn

GetDone:        cmp     ax, 0                   ;Check for EOF
                je      AtEOF

                print
                byte    "Error copying files (read error)",cr,lf,0

AtEOF:          lesi    OutFile
                fclose
CloseIn:        lesi    InFile
                fclose

Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack   ")
sseg            ends
```

```
zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

13.7 Programming Projects

1) The sample program in Section 13.5 reroutes the standard output through the Standard Library's file I/O routines allowing you to use any of the output routines to write data to a file. Write a similar set of routines and macros that let you read data from a file using the Standard Library's input routines (getc, gets, getsm scanf, etc.). Redirect the input through the Standard Library's file input functions.

2) The last sample program in section 13.3.12 (copyuc.asm on the companion CD-ROM) copies one file to another, possibly converting lower case characters to upper case. This program currently parses the command line directly and uses blocked I/O to copy the data in the file. Rewrite this program using argv/argc to process the command line parameters and use the Standard Library file I/O routines to process each character in the file.

3) Write a "word count" program that counts the number of characters, words, and lines within a file. Assume that a word is any sequence of characters between spaces, tabs, carriage returns, line feeds, the beginning of a file, and the end of a file (if you want to save some effort, you can assume a "whitespace" symbol is any ASCII code less than or equal to a space).

4) Write a program that prints an ASCII text file to the printer. Use the BIOS int 17h services to print the characters in the file.

5) Write two programs, "xmit" and "rcv". The xmit program should fetch a command line filename and transmit this file across the serial port. It should transmit the filename and the number of bytes in the file (hint: use the DOS seek command to determine the length of the file). The rcv program should read the filename and file length from the serial port, create the file by the specified name, read the specified number of bytes from the serial port, and then close the file.

13.8 Summary

MS-DOS and BIOS provide many system services which control the hardware on a PC. They provide a machine independent and flexible interface. Unfortunately, the PC has grown up quite a bit since the days of the original 5 Mhz 8088 IBM PC. Many BIOS and DOS calls are now obsolete, having been superseded by newer calls. To ensure backwards compatibility, MS-DOS and BIOS generally support all of the older obsolete calls as well as the newer calls. However, your programs should not use the obsolete calls, they are there for backwards compatibility only.

The BIOS provides many services related to the control of devices such as the video display, the printer port, the keyboard, the serial port, the real time clock, etc. Descriptions of the BIOS services for these devices appear in the following sections:

- "INT 5- Print Screen" on page 702
- "INT 10h - Video Services" on page 702
- "INT 11h - Equipment Installed" on page 704
- "INT 12h - Memory Available" on page 704
- "INT 13h - Low Level Disk Services" on page 704
- "INT 14h - Serial I/O" on page 706
- "INT 15h - Miscellaneous Services" on page 708
- "INT 16h - Keyboard Services" on page 708
- "INT 17h - Printer Services" on page 710
- "INT 18h - Run BASIC" on page 712
- "INT 19h - Reboot Computer" on page 712

- “INT 1Ah - Real Time Clock” on page 712

MS-DOS provides several different types of services. This chapter concentrated on the file I/O services provided by MS-DOS. In particular, this chapter dealt with implementing efficient file I/O operations using blocked I/O. To learn how to perform file I/O and perform other MS-DOS operations, check out the following sections:

- “MS-DOS Calling Sequence” on page 714
- “MS-DOS Character Oriented Functions” on page 714
- “MS-DOS “Obsolete” Filing Calls” on page 717
- “MS-DOS Date and Time Functions” on page 718
- “MS-DOS Memory Management Functions” on page 718
- “MS-DOS Process Control Functions” on page 721
- “MS-DOS “New” Filing Calls” on page 725
- “File I/O Examples” on page 734
- “Blocked File I/O” on page 737

Accessing command line parameters is an important operation within MS-DOS applications. DOS’ PSP (Program Segment Prefix) contains the command line and several other pieces of important information. To learn about the various fields in the PSP and see how to access command line parameters, check out the following sections in this chapter:

- “The Program Segment Prefix (PSP)” on page 739
- “Accessing Command Line Parameters” on page 742
- “ARGC and ARGV” on page 750

Of course, the UCR Standard Library provides some file I/O routines as well. This chapter closes up by describing some of the StdLib file I/O routines along with their advantages and disadvantages. See

- “Fopen” on page 751
- “Fcreate” on page 752
- “Fclose” on page 752
- “Fflush” on page 752
- “Fgetc” on page 752\
- “Fread” on page 753
- “Fputc” on page 753
- “Fwrite” on page 753
- “Redirecting I/O Through the StdLib File I/O Routines” on page 753
- “A File I/O Example” on page 755

13.9 Questions

1) How are BIOS routines called?

2) Which BIOS routine is used to write a character to the:

   a) video display b) serial port c) printer port

3) When the serial transmit or receive services return to the caller, the error status is returned in the AH register. However, there is a problem with the value returned. What is this problem?

4) Explain how you could test the keyboard to see if a key is available. 5)What is wrong with the keyboard shift status function?

6) How are special key codes (those keystrokes not returning ASCII codes) returned by the read keyboard call?

7) How would you send a character to the printer?

8) How do you read the real time clock?

9) Given that the RTC increments a 32-bit counter every 55ms, how long will the system run before overflow of this counter occurs?

10) Why should you reset the clock if, when reading the clock, you've determined that the counter has overflowed?

11) How do assembly language programs call MS-DOS?

12) Where are parameters generally passed to MS-DOS?

13) Why are there two sets of filing functions in MS-DOS?

14) Where can the DOS command line be found?

15) What is the purpose of the environment string area?

16) How can you determine the amount of memory available for use by your program?

17) Which is more efficient: character I/O or blocked I/O? Why?

18) What is a good blocksize for blocked I/O?

19) What can't you use blocked I/O on random access files?

20) Explain how to use the seek command to move the file pointer 128 bytes backwards in the file from the current file position.

21) Where is the error status normally returned after a call to DOS?

22) Why is it difficult to use blocked I/O on a random access file? Which would be easier, random access on a blocked I/O file opened for input or random access on a blocked I/O file opened for reading and writing?

23) Describe how you might implement blocked I/O on files opened for random access reading and writing.

24) What are two ways you can obtain the address of the PSP?

25) How do you determine that you've reached the end of file when using MS-DOS file I/O calls? When using UCR Standard Library file I/O calls?

Floating Point Arithmetic Chapter 14

Although integers provide an exact representation for numeric values, they suffer from two major drawbacks: the inability to represent fractional values and a limited dynamic range. Floating point arithmetic solves these two problems at the expense of accuracy and, on some processors, speed. Most programmers are aware of the speed loss associated with floating point arithmetic; however, they are blithely unware of the problems with accuracy.

For many applications, the benefits of floating point outweigh the disadvantages. However, to properly use floating point arithmetic in *any* program, you must learn how floating point arithmetic operates. Intel, understanding the importance of floating point arithmetic in modern programs, provided support for floating point arithmetic in the earliest designs of the 8086 – the 80x87 FPU (floating point unit or math coprocessor). However, on processors eariler than the 80486 (or on the 80486sx), the floating point processor is an optional device; it this device is not present you must simulate it in software.

This chapter contains four main sections. The first section discusses floating point arithmetic from a mathematical point of view. The second section discusses the binary floating point formats commonly used on Intel processors. The third discusses software floating point and the math routines from the UCR Standard Library. The fourth section discusses the 80x87 FPU chips.

14.0 Chapter Overview

This chapter contains four major sections: a description of floating point formats and operations (two sections), a discussion of the floating point support in the UCR Standard Library, and a discussion of the 80x87 FPU (floating point unit). The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- The mathematics of floating point arithmetic.
- IEEE floating point formats.
- The UCR Standard Library floating point routines.
- The 80x87 floating point coprocessors.
- FPU data movement instructions.
- ❑ Conversions.
- Arithmetic instructions.
- Comparison instructions.
- ❑ Constant instructiuons.
- ❑ Transcendental instructions.
- ❑ Miscellaneous instructions.
- ❑ Integer operations.
- ❑ Additional trigonometric functions.

14.1 The Mathematics of Floating Point Arithmetic

A big problem with floating point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating point arithmetic. This is a source of bugs in many programs. One of the primary goals of this section is to describe the limitations of floating point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinte precision* arithmetic. Consider the simple statement x:=x+1, x is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur.* That is, this statement is valid only for

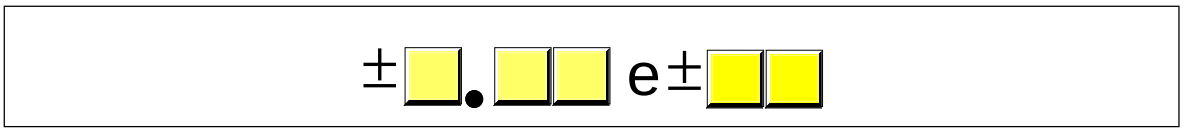


Figure 14.1 Simple Floating Point Format

certain values of x (minint <= x < maxint). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules (e.g., $5 / 2 \neq 2.5$).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating point values suffer from this same problem, only worse. After all, the integers are a subset of the real numbers. Therefore, the floating point values must represent the same infinite set of integers. However, there are an infinite number of values between any two real values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges, either.

To represent real numbers, most floating point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating point arithmetic operations. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating point format for our examples. Our floating point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values (see Figure 14.1).

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, you must adjust the values so they have the same exponent. One way to do this is to to convert 4.56e0 to 0.456e1 and then add. This produces 1.686e1. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain 1.69e1. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating point calculation is limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining 1.68e1 which is even less correct. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value which is the result of a sequence of floating point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add 1.23e3 with 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable to you; after all, we can only maintain three significant digits, adding in a small value shouldn't affect the result at all. However, suppose we were to add 1.00e0 1.23e3 *ten times*. The first time we add 1.00e0 to 1.23e3 we get 1.23e3. Likewise, we get this same result the second, third, fourth, ..., and tenth time we add 1.00e0 to 1.23e3. On the other hand, had we added 1.00e0 to itself ten times, then added the result (1.00e1) to 1.23e3, we would have gotten a different result, 1.24e3. This is the most important thing to know about limited precision arithmetic:

The order of evaluation can effect the accuracy of the result.

You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation 1.23e0 - 1.22 e0. This produces 0.01e0. Although this is mathematically equivalent to 1.00e-2, this latter form suggests that the last two digits are exactly zero. Unfortunately, we've only got a single significant digit at this time. Indeed, some FPUs or floating point software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

> *Whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating point format.*

Multiplication and division do not suffer from the same problems as addition and subtraction since you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error which already exists in a value. For example, if you multiply 1.23e0 by two, when you should be multiplying 1.24e0 by two, the result is even less accurate. This brings up a third important rule when working with limited precision arithmetic:

> *When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.*

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute x*(y+z). Normally you would add y and z together and multiply their sum by x. However, you will get a little more accuracy if you transform x*(y+z) to get x*y+x*z and compute the result by performing the multiplications first.

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

> *When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.*

Comparing floating pointer numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating point value), you should *never* compare two floating point values to see if they are equal. In a binary floating point format, different computations which produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0+1.69e0 should produce 3.00e0. Likewise, adding 2.50e0+1.50e0 should produce 3.00e0. However, were you to compare (1.31e0+1.69e0) agains (2.50e0+1.50e0) you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Since this is not necessarily true after two different floating point computations which should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The straight-forward way to do this is to use a test like the following:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then …
```

Another common way to handle this same comparison is to use a statement of the form:

```
if abs(Value1-Value2) <= error then …
```

Most texts, when discussing floating point comparisons, stop immediately after discussing the problem with floating point equality, assuming that other forms of comparison are perfectly okay with floating point numbers. This isn't true! If we are assuming that x=y if x is within y±error, then a simple bitwise comparison of x and y will claim that x<y if y is greater than x but less than y+error. However, in such a case x should really be treated as equal to y, not less than y. Therefore, we must always compare two floating point numbers using ranges, regardless of the actual comparison we want to perform. Trying to compare two floating point numbers directly can lead to an error. To compare two floating point numbers, x and y, against one another, you should use one of the following forms:

```
=         if abs(x-y) <= error then …
≠         if abs(x-y) > error then …
<         if (x-y) < error then …
≤         if (x-y) <= error then …
>         if (x-y) > error then …
≥         if (x-y) >= error then …
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error which will creep into your computations. The exact value will depend upon the particular floating point format you use, but more on that a little later. The final rule we will state in this section is

> *When comparing two floating point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.*

There are many other little problems that can occur when using floating point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating point arithmetic like real arithmetic – the inaccuracies present in limited precision arithmetic can get you into trouble if you are not careful. A good text on numerical analysis or even scientific computing can help fill in the details which are beyond the scope of this text. If you are going to be working with floating point arithmetic, *in any language*, you should take the time to study the effects of limited precision arithmetic on your computations.

14.2 IEEE Floating Point Formats

When Intel planned to introduce a floating point coprocessor for their new 8086 microprocessor, they were smart enough to realize that the electrical engineers and solid-state physicists who design chips were, perhaps, not the best people to do the necessary numerical analysis to pick the best possible binary representation for a floating point format. So Intel went out and hired the best numerical analyst they could find to design a floating point format for their 8087 FPU. That person then hired two other experts in the field and the three of them (Kahn, Coonan, and Stone) designed Intel's floating point format. They did such a good job designing the KCS Floating Point Standard that the IEEE organization adopted this format for the IEEE floating point format[1].

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating point formats: single precision, double precision, and extended precision. The single and double precision formats corresponded to C's float and double types or FORTRAN's real and double precision types. Intel intended to use extended precision for long chains of computations. Extended precision contains 16 extra bits that the

1. There were some minor changes to the way certain degenerate operations were handled, but the bit representation remained essentially unchanged.

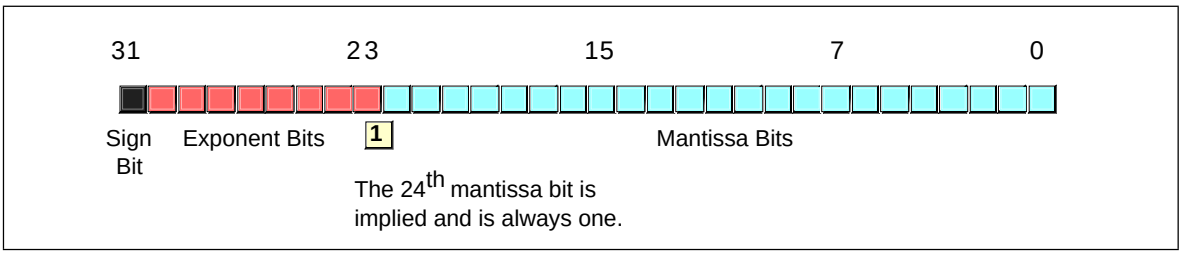


Figure 14.2 32 Bit Single Precision Floating Point Format

calculations could use for guard bits before rounding down to a double precision value when storing the result.

The single precision format uses a one's complement 24 bit mantissa and an eight bit excess-128 exponent. The mantissa usually represents a value between 1.0 to just under 2.0. The H.O. bit of the mantissa is always assumed to be one and represents a value just to the left of the *binary point*[2]. The remaining 23 mantissa bits appear to the right of the binary point. Therefore, the mantissa represents the value:

```
1.mmmmmmm mmmmmmmm mmmmmmmm
```

The "mmmm…" characters represent the 23 bits of the mantissa. Keep in mind that we are working with binary numbers here. Therefore, each position to the right of the binary point represents a value (zero or one) times a successive negative power of two. The implied one bit is always multiplied by 2^0, which is one. This is why the mantissa is always greater than or equal to one. Even if the other mantissa bits are all zero, the implied one bit always gives us the value one[3]. Of course, even if we had an almost infinite number of one bits after the binary point, they still would not add up to two. This is why the mantissa can represent values in the range one to just under two.

Although there are an infinite number of values between one and two, we can only represent eight million of them because we a 23 bit mantissa (the 24th bit is always one). This is the reason for inaccuracy in floating point arithmetic – we are limited to 23 bits of precision in compuations involving single precision floating point values.

The mantissa uses a *one's complement* format rather than two's complement. This means that the 24 bit value of the mantissa is simply an unsigned binary number and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for zero (with the sign bit set or clear). Generally, this is important only to the person designing the floating point software or hardware system. We will assume that the value zero always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating point format comes into play. The floating point format raise two to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is eight bits and is stored in an *excess-127* format. In excess-127 format, the exponent 2^0 is represented by the value 127 (7fh). Therefore, to convert an exponent to excess-127 format simply add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating point values. The single precision floating point format takes the form shown in Figure 14.2.

With a 24 bit mantissa, you will get approximately $6\text{-}^1/_2$ digits of precision (one half digit of precision means that the first six digits can all be in the range 0..9 but the seventh digit can only be in the range 0..x where x<9 and is generally close to five). With an eight

2. The binary point is the same thing as the decimal point except it appears in binary numbers rather than decimal numbers.

3. Actually, this isn't necessarily true. Thye IEEE floating point format supports *denormalized* values where the H.O. bit is not zero. However, we will ignore denormalized values in our discussion.

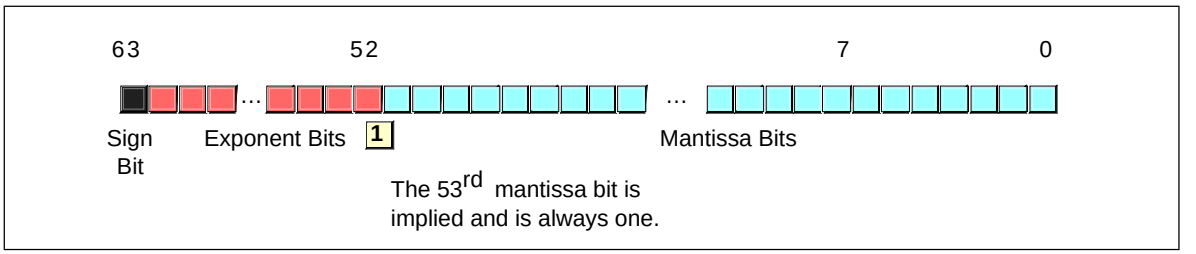


Figure 14.3 64 Bit Double Precision Floating Point Format

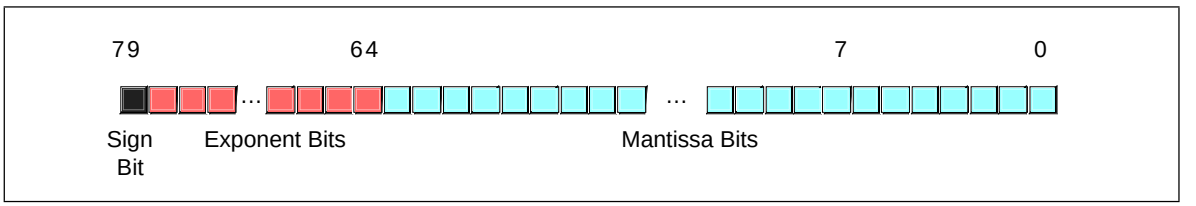


Figure 14.4 80 Bit Extended Precision Floating Point Format

bit excess-128 exponent, the dynamic range of single precision floating point numbers is approximately $2^{\pm 128}$ or about $10^{\pm 38}$.

Although single precision floating point numbers are perfectly suitable for many applications, the dynamic range is somewhat small for many scientific applications and the very limited precision is unsuitable for many financial, scientific, and other applications. Furthermore, in long chains of computations, the limited precision of the single precision format may introduce serious error.

The double precision format helps overcome the problems of single preicision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa (with an implied H.O. bit of one) plus a sign bit. This provides a dynamic range of about $10^{\pm 308}$and 14-$^{1}/_{2}$ digits of precision, sufficient for most applications. Double precision floating point values take the form shown in Figure 14.3.

In order to help ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa, four of the additional bits are appended to the end of the exponent. Unlike the single and double precision values, the extended precision format does not have an implied H.O. bit which is always one. Therefore, the extended precision format provides a 64 bit mantissa, a 15 bit excess-16383 exponent, and a one bit sign. The format for the extended precision floating point value is shown in Figure 14.4.

On the 80x87 FPUs and the 80486 CPU, all computations are done using the extended precision form. Whenever you load a single or double precision value, the FPU automatically converts it to an extended precision value. Likewise, when you store a single or double precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. By always working with the extended precision format, Intel guarantees a large number of guard bits are present to ensure the accuracy of your computations. Some texts erroneously claim that you should never use the extended precision format in your own programs, because Intel only guarantees accurate computations when using the single or double precision formats. This is foolish. By performing all computations using 80 bits, Intel helps ensure (but not guarantee) that you will get full 32 or 64 bit accuracy in your computations. Since the 80x87 FPUs and 80486 CPU do not provide a large number of guard bits in 80 bit computations, some error will inevitably creep into the L.O. bits of an extended precision computation. However, if your computation is correct to 64 bits, the 80 bit computation will always provide *at least* 64 accurate bits. Most of the time you will get even more. While you cannot assume that you get an accurate 80

bit computation, you can usually do better than 64 when using the extended precision format.

To maintain maximum precision during computation, most computations use *normalized* values. A normalized floating point value is one that has a H.O. mantissa bit equal to one. Almost any non-normalized value can be normalized by shifting the mantissa bits to the left and decrementing the exponent by one until a one appears in the H.O. bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating point value by two. Likewise, whenever you decrement the exponent, you divide the floating point value by two. By the same token, shifting the mantissa to the left one bit position multiplies the floating point value by two; likewise, shifting the mantissa to the right divides the floating point value by two. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating point number at all.

Keeping floating point numbers normalized is beneficial because it maintains the maximum number of bits of precision for a computation. If the H.O. bits of the mantissa are all zero, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating point computation will be more accurate if it involves only normalized values.

There are two important cases where a floating point number cannot be normalized. The value 0.0 is a special case. Obviously it cannot be normalized because the floating point representation for zero has no one bits in the mantissa. This, however, is not a problem since we can exactly represent the value zero with only a single bit.

The second case is when we have some H.O. bits in the mantissa which are zero but the biased exponent is also zero (and we cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose H.O. mantissa bits and biased exponent are zero (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values[4]. Although the use of denormalized values allows IEEE floating point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer less bits of precision and are inherently less accurate.

Since the 80x87 FPUs and 80486 CPU always convert single and double precision values to extended precision, extended precision arithmetic is actually *faster* than single or double precision. Therefore, the expected performance benefit of using the smaller formats is not present on these chips. However, when designing the Pentium / 586 CPU, Intel redesigned the built-in floating point unit to better compete with RISC chips. Most RISC chips support a native 64 bit double precision format which is faster than Intel's extended precision format. Therefore, Intel provided native 64 bit operations on the Pentium to better compete against the RISC chips. Therefore, the double precision format is the fastest on the Pentium and later chips.

14.3 The UCR Standard Library Floating Point Routines

In most assembly language texts, which bother to cover floating point arithmetic, this section would normally describe how to design your own floating point routines for addition, subtraction, multiplication, and division. This text will not do that for several reasons. First, to design a *good* floating point library requires a solid background in numerical analysis; a prerequisite this text does not assume of its readers. Second, the UCR Standard Library already provides a reasonable set of floating point routines in source code form; why waste space in this text when the sources are readily available elsewhere? Third, floating point units are quickly becoming standard equipment on all modern CPUs or motherboards; it makes no more sense to describe how to manually perform a floating point computation than it does to describe how to manually perform an integer computation. Therefore, this section will describe how to use the UCR Standard Library routines if

4. The alternative would be to underflow the values to zero.

you do not have an FPU available; a later section will describe the use of the floating point unit.

The UCR Standard Library provides a large number of routines to support floating point computation and I/O. This library uses the same memory format for 32, 64, and 80 bit floating point numbers as the 80x87 FPUs. The UCR Standard Library's floating point routines do not exactly follow the IEEE requirements with respect to error conditions and other degenerate cases, and it may produce slightly different results than an 80x87 FPU, but the results will be very close[5]. Since the UCR Standard Library uses the same memory format for 32, 64, and 80 bit numbers as the 80x87 FPUs, you can freely mix computations involving floating point between the FPU and the Standard Library routines.

The UCR Standard Library provides numerous routines to manipulate floating point numbes. The following sections describe each of these routines, by category.

14.3.1 Load and Store Routines

Since 80x86 CPUs without an FPU do not provide any 80-bit registers, the UCR Standard Library must use memory-based variables to hold floating point values during computation. The UCR Standard Library routines use two *pseudo registers*, an accumlator register and an operand register, when performing floating point operations. For example, the floating point addition routine adds the value in the floating point operand register to the floating point accumulator register, leaving the result in the accumulator. The load and store routines allow you to load floating point values into the floating point accumulator and operand registers as well as store the value of the floating point accumulator back to memory. The routines in this category include accop, xaccop, lsfpa, ssfpa, ldfpa, sdfpa, lefpa, sefpa, lefpal, lsfpo, ldfpo, lefpo, and lefpol.

The accop routine copies the value in the floating point accumulator to the floating point operand register. This routine is useful when you want to use the result of one computation as the second operand of a second computation.

The xaccop routine exchanges the values in the floating point accumuator and operand registers. Note that many floating point computations destory the value in the floating point operand register, so you cannot blindly assume that the routines preserve the operand register. Therefore, calling this routine only makes sense after performing some computation which you know does not affect the floating point operand register.

Lsfpa, ldfpa, and lefpa load the floating point accumulator with a single, double, or extended precision floating point value, respectively. The UCR Standard Library uses its own internal format for computations. These routines convert the specified values to the internal format during the load. On entry to each of these routines, es:di must contain the address of the variable you want to load into the floating point accumulator. The following code demonstrates how to call these routines:

```
rVar            real4    1.0
drVar           real8    2.0
xrVar           real10   3.0
                  .
                  .
                  .
                lesi     rVar
                lsfpa
                  .
                  .
                  .
                lesi     drVar
                ldfpa
                  .
                  .
                  .
```

5. Note, by the way, that different floating point chips, especially across different CPU lines, but even within the Intel family, produce slightly different results. So the fact that the UCR Standard Library does not produce the exact same results as a particular FPU is not that important.

```
                lesi    xrVar
                lefpa
```

The lsfpo, ldfpo, and lefpo routines are similar to the lsfpa, ldfpa, and lefpa routines except, of course, they load the floating point operand register rather than the floating point accumulator with the value at address es:di.

Lefpal and lefpol load the floating point accumulator or operand register with a literal 80 bit floating point constant appearing in the code stream. To use these two routines, simply follow the call with a real10 directive and the appropriate constant, e.g.,

```
                lefpal
                real10  1.0
                lefpol
                real10  2.0e5
```

The ssfpa, sdfpa, and sefpa routines store the value in the floating point accumulator into the memory based floating point variable whose address appears in es:di. There are no corresponding ssfpo, sdfpo, or sefpo routines because a result you would want to store should never appear in the floating point operand register. If you happen to get a value in the floating point operand that you want to store into memory, simply use the xaccop routine to swap the accumulator and operand registers, then use the store accumulator routines to save the result. The following code demonstrates the use of these routines:

```
rVar            real4   1.0
drVar           real8   2.0
xrVar           real10  3.0
                  .
                  .
                  .
                lesi    rVar
                ssfpa
                  .
                  .
                  .
                lesi    drVar
                sdfpa
                  .
                  .
                  .
                lesi    xrVar
                sefpa
```

14.3.2 Integer/Floating Point Conversion

The UCR Standard Library includes several routines to convert between binary integers and floating point values. These routines are itof, utof, ltof, ultof, ftoi, ftou, ftol, and ftoul. The first four routines convert signed and unsigned integers to floating point format, the last four routines truncate floating point values and convert them to an integer value.

Itof converts the signed 16-bit value in ax to a floating point value and leaves the result in the floating point accumulator. This routine does not affect the floating point operand register. Utof converts the unsigned integer in ax in a similar fashion. Ltof and ultof convert the 32 bit signed (ltof) or unsigned (ultof) integer in dx:ax to a floating point value, leaving the value in the floating point accumulator. These routines always succeed.

Ftoi converts the value in the floating point accumulator to a signed integer value, leaving the result in ax. Conversion is by truncation; this routine keeps the integer portion and throws away the fractional part. If an overflow occurs because the resulting integer portion does not fit into 16 bits, ftoi returns the carry flag set. If the conversion occurs without error, ftoi return the carry flag clear. Ftou works in a similar fashion, except it converts the floating point value to an unsigned integer in ax; it returns the carry set if the floating point value was negative.

Ftol and ftoul converts the value in the floating point accumulator to a 32 bit integer leaving the result in dx:ax. Ftol works on signed values, ftoul works with unsigned values. As with ftoi and ftou, these routines return the carry flag set if a conversion error occurs.

14.3.3 Floating Point Arithmetic

Floating point arithmetic is handled by the fpadd, fp sub, fpcmp, fpmul, and fpdiv routines. Fpadd adds the value in the floating point accumulator to the floating point accumulator. Fpsub subtracts the value in the floating point operand from the floating point accumulator. Fpmul multiplies the value in the floating accumulator by the floating point operand. Fpdiv divides the value in the floating point accumulator by the value in the floating point operand register. Fpcmp compares the value in the floating point accumulator against the floating point operand.

The UCR Standard Library arithmetic routines do very little error checking. For example, if arithmetic overflow occurs during addition, subtraction, multiplication, or division, the Standard Library simply sets the result to the largest legal value and returns. This is one of the major deviations from the IEEE floating point standard. Likewise, when underflow occurs the routines simply set the result to zero and return. If you divide any value by zero, the Standard Library routines simply set the result to the largest possible value and return. You may need to modify the standard library routines if you need to check for overflow, underflow, or division by zero in your programs.

The floating point comparison routine (fpcmp) compares the floating point accumulator against the floating point operand and returns -1, 0, or 1 in the ax register if the accumulator is less than, equal, or greater than the floating point operand. It also compares ax with zero immediately before returning so it sets the flags so you can use the jg, jge, jl, jle, je, and jne instructions immediately after calling fpcmp. Unlike fpadd, fpsub, fpmul, and fpdiv, fpcmp does not destroy the value in the floating point accumulator or the floating point operand register. Keep in mind the problems associated with comparing floating point numbers!

14.3.4 Float/Text Conversion and Printff

The UCR Standard Library provides three routines, ftoa, etoa, and atof, that let you convert floating point numbers to ASCII strings and vice versa; it also provides a special version of printf, printff, that includes the ability to print floating point values as well as other data types.

Ftoa converts a floating point number to an ASCII string which is a decimal representation of that floating point number. On entry, the floating point accumulator contains the number you want to convert to a string. The es:di register pair points at a buffer in memory where ftoa will store the string. The al register contains the field width (number of print positions). The ah register contains the number of positions to display to the right of the decimal point. If ftoa cannot display the number using the print format specified by al and ah, it will create a string of "#" characters, ah characters long. Es:di must point at a byte array containing at least al+1 characters and al should contain at least five. The field width and decimal length values in the al and ah registers are similar to the values appearing after floating point numbers in the Pascal write statement, e.g.,

```
write(floatVal:al:ah);
```

Etoa outputs the floating point number in exponential form. As with ftoa, es:di points at the buffer where etoa will store the result. The al register must contain at least eight and is the field width for the number. If al contains less than eight, etoa will output a string of "#" characters. The string that es:di points at must contain at least al+1 characters. This conversion routine is similar to Pascal's write procedure when writing real values with a single field width specification:

```
write(realvar:al);
```

The Standard Library printff routine provides all the facilities of the standard printf routine plus the ability to handle floating point output. The printff routine includes sev-

eral new format specifications to print floating point numbers in decimal form or using scientific notation. The specifications are

- %x.yF Prints a 32 bit floating point number in decimal form.
- %x.yGF Prints a 64 bit floating point number in decimal form.
- %x.yLF Prints an 80 bit floating point number in decimal form.
- %zE Prints a 32 bit floating point number using scientific notation.
- %zGE Prints a 64 bit floating point number using scientific notation.
- %zLE Prints an 80 bit floating point value using scientific notation.

In the format strings above, *x* and *z* are integer constants that denote the field width of the number to print. The *y* item is also an integer constant that specifies the number of positions to print after the decimal point. The *x.y* values are comparable to the values passed to ftoa in al and ah. The *z* value is comparable to the value etoa expects in the al register.

Other than the addition of these six new formats, the printff routine is identical to the printf routine. If you use the printff routine in your assembly language programs, you should *not* use the printf routine as well. Printff duplicates all the facilities of printf and using both would only waste memory.

14.4 The 80x87 Floating Point Coprocessors

When the 8086 CPU first appeared in the late 1970's, semiconductor technology was not to the point where Intel could put floating point instrutions directly on the 8086 CPU. Therefore, they devised a scheme whereby they could use a second chip to perform the floating point calculations – the floating point unit (or FPU)[6]. They released their original floating point chip, the 8087, in 1980. This particular FPU worked with the 8086, 8088, 80186, and 80188 CPUs. When Intel introduced the 80286 CPU, they released a redesigned 80287 FPU chip to accompany it. Although the 80287 was compatible with the 80386 CPU, Intel designed a better FPU, the 80387, for use in 80386 systems. The 80486 CPU was the first Intel CPU to include an on-chip floating point unit. Shortly after the release of the 80486, Intel introduced the 80486sx CPU that was an 80486 without the built-in FPU. To get floating point capabilities on this chip, you had to add an 80487 chip, although the 80487 was really nothing more than a full-blown 80486 which took over for the "sx" chip in the system. Intel's Pentium/586 chips provide a high-performance floating point unit directly on the CPU. There is no floating point coprocessor available for the Pentium chip.

Collectively, we will refer to all these chips as the 80x87 FPU. Given the obsolesence of the 8086, 80286, 8087, and 80287 chips, this text will concentrate on the 80387 and later chips. There are some differences between the 80387/80486/Pentium floating point units and the earlier FPUs. If you need to write code that will execute on those earlier machines, you should consult the appropriate Intel documentation for those devices.

14.4.1 FPU Registers

The 80x87 FPUs add 13 registers to the 80386 and later processors: eight floating point data registers, a control register, a status register, a tag register, an instruction pointer, and a data pointer. The data registers are similar to the 80x86's general purpose register set insofar as all floating point calculations take place in these registers. The control register contains bits that let you decide how the 80x87 handles certain degenerate cases like rounding of inaccurate computations, control precision, and so on. The status register is similar to the 80x86's flags register; it contains the condition code bits and several other floating point flags that describe the state of the 80x87 chip. The tag register contains several groups of bits that determine the state of the value in each of the eight general purpose registers. The instruction and data pointer registers contain certain state information

6. Intel has also refered to this device as the Numeric Data Processor (NDP), Numeric Processor Extension (NPX), and math coprocessor.

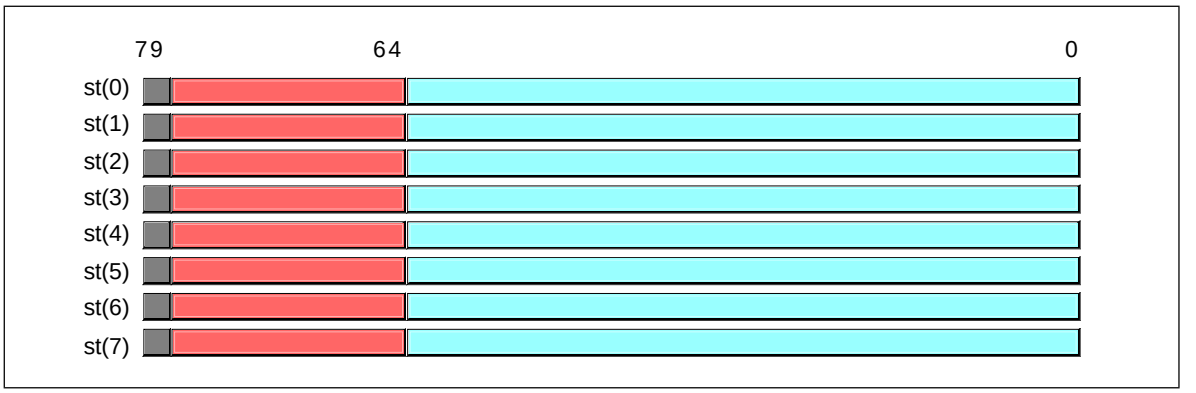


Figure 14.5 80x87 Floating Point Register Stack

about the last floating point instruction executed. We will not consider the last three registers in this text, see the Intel documentation for more details.

14.4.1.1 The FPU Data Registers

The 80x87 FPUs provide eight 80 bit data registers organized as a stack. This is a significant departure from the organization of the general purpose registers on the 80x86 CPU that comprise a standard general-purpose register set. Intel refers to these registers as ST(0), ST(1), …, ST(7). Most assemblers will accept ST as an abbreviation for ST(0).

The biggest difference between the FPU register set and the 80x86 register set is the stack organization. On the 80x86 CPU, the ax register is always the ax register, no matter what happens. On the 80x87, however, the register set is an eight element stack of 80 bit floating point values (see Figure 14.5). ST(0) refers to the item on the top of the stack, ST(1) refers to the next item on the stack, and so on. Many floating point instructions push and pop items on the stack; therefore, ST(1) will refer to the previous contents of ST(0) after you push something onto the stack. It will take some thought and practice to get used to the fact that the registers are changing under you, but this is an easy problem to overcome.

14.4.1.2 The FPU Control Register

When Intel designed the 80x87 (and, essentially, the IEEE floating point standard), there were no standards in floating point hardware. Different (mainframe and mini) computer manufacturers all had different and incompatible floating point formats. Unfortunately, much application software had been written taking into account the idiosyncrasies of these different floating point formats. Intel wanted to designed an FPU that could work with the majority of the software out there (keep in mind, the IBM PC was three to four years away when Intel began designing the 8087, they couldn't rely on that "mountain" of software available for the PC to make their chip popular). Unfortunately, many of the features found in these older floating point formats were mutually exclusive. For example, in some floating point systems rounding would occur when there was insufficient precision; in others, truncation would occur. Some applications would work with one floating point system but not with the other. Intel wanted as many applications as possible to work with as few changes as possible on their 80x87 FPUs, so they added a special register, the FPU *control register*, that lets the user choose one of several possible operating modes for the 80x87.

The 80x87 control register contains 16 bits organized as shown in Figure 14.6.

Bit 12 of the control register is only present on the 8087 and 80287 chips. It controls how the 80x87 responds to infinity. The 80387 and later chips always use a form of infinitly known and *affine closure* because this is the only form supported by the IEEE

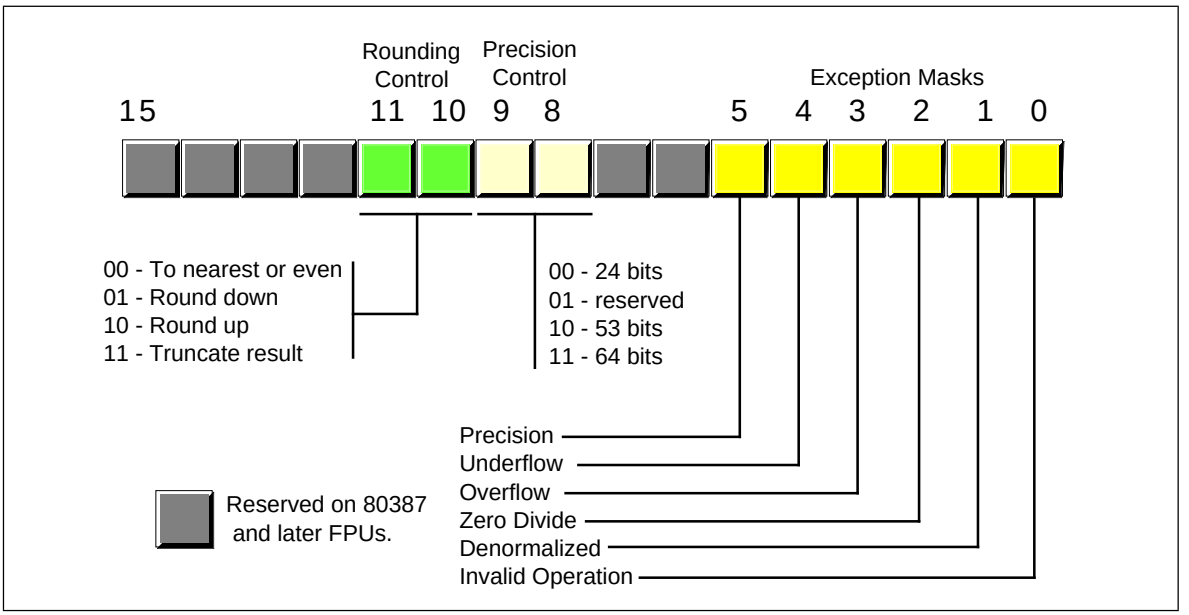


Figure 14.6 80x87 Control Register

754/854 standards. As such, we will ignore any further use of this bit and assume that it is always programmed with a one.

Bits 10 and 11 provide rounding control according to the following values:

Table 58: Rounding Control

Bits 10 & 11	Function
00	To nearest or even
01	Round down
10	Round up
11	Truncate

The “00” setting is the default. The 80x87 rounds values above one-half of the least significant bit up. It rounds values below one-half of the least significant bit down. If the value below the least significant bit is exactly one-half the least significant bit, the 80x87 rounds the value towards the value whose least significant bit is zero. For long strings of computations, this provides a reasonable, automatic, way to maintain maximum precision.

The round up and round down options are present for those computations where it is important to keep track of the accuracy during a computation. By setting the rounding control to round down and performing the operation, the repeating the operation with the rounding control set to round up, you can determine the minimum and maximum ranges between which the true result will fall.

The truncate option forces all computations to truncate any excess bits during the computation. You will rarely use this option if accuracy is important to you. However, if you are porting older software to the 80x87, you might use this option to help when porting the software.

Bits eight and nine of the control register control the precision during computation. This capability is provided mainly to allow compatbility with older software as required by the IEEE 754 standard. The precision control bits use the following values:

Table 59: Mantissa Precision Control Bits

Bits 8 & 9	Precision Control
00	24 bits
01	Reserved
10	53 bits
11	64 bits

For modern applications, the precision control bits should always be set to “11” to obtain 64 bits of precision. This will produce the most accurate results during numerical computation.

Bits zero through five are the *exception masks*. These are similar to the interrupt enable bit in the 80x86’s flags register. If these bits contain a one, the corresponding condition is ignored by the 80x87 FPU. However, if any bit contains zero, and the corresponding condition occurs, then the FPU immediately generates an interrupt so the program can handle the degenerate condition.

Bit zero corresponds to an invalid operation error. This generally occurs as the result of a programming error. Problem which raise the invalid operation exception include pushing more than eight items onto the stack or attempting to pop an item off an empty stack, taking the square root of a negative number, or loading a non-empty register.

Bit one masks the *denormalized* interrupt which occurs whenever you try to manipulate denormalized values. Denormalized values generally occur when you load arbitrary extended precision values into the FPU or work with very small numbers just beyond the range of the FPU’s capabilities. Normally, you would probably *not* enable this exception.

Bit two masks the *zero divide* exception. If this bit contains zero, the FPU will generate an interrupt if you attempt to divide a nonzero value by zero. If you do not enable the zero division exception, the FPU will produce NaN (not a number) whenever you perform a zero division.

Bit three masks the overflow exception. The FPU will raise the overflow exception if a calculation overflows or if you attempt to store a value which is too large to fit into a destination operand (e.g., storing a large extended precision value into a single precision variable).

Bit four, if set, masks the *underflow* exception. Underflow occurs when the result is too *small* to fit in the desintation operand. Like overflow, this exception can occur whenever you store a small extended precision value into a smaller variable (single or double precision) or when the result of a computation is too small for extended precision.

Bit five controls whether the *precision* exception can occur. A precision exception occurs whenever the FPU produces an imprecise result, generally the result of an internal rounding operation. Although many operations will produce an exact result, many more will not. For example, dividing one by ten will produce an inexact result. Therefore, this bit is usually one since inexact results are very common.

Bits six and thirteen through fifteen in the control register are currently undefined and reserved for future use. Bit seven is the interrupt enable mask, but it is only active on the 8087 FPU; a zero in this bit enables 8087 interrupts and a one disables FPU interrupts.

The 80x87 provides two instructions, FLDCW (load control word) and FSTCW (store control word), that let you load and store the contents of the control register. The single operand to these instructions must be a 16 bit memory location. The FLDCW instruction loads the control register from the specified memory location, FSTCW stores the control register into the specified memory location.

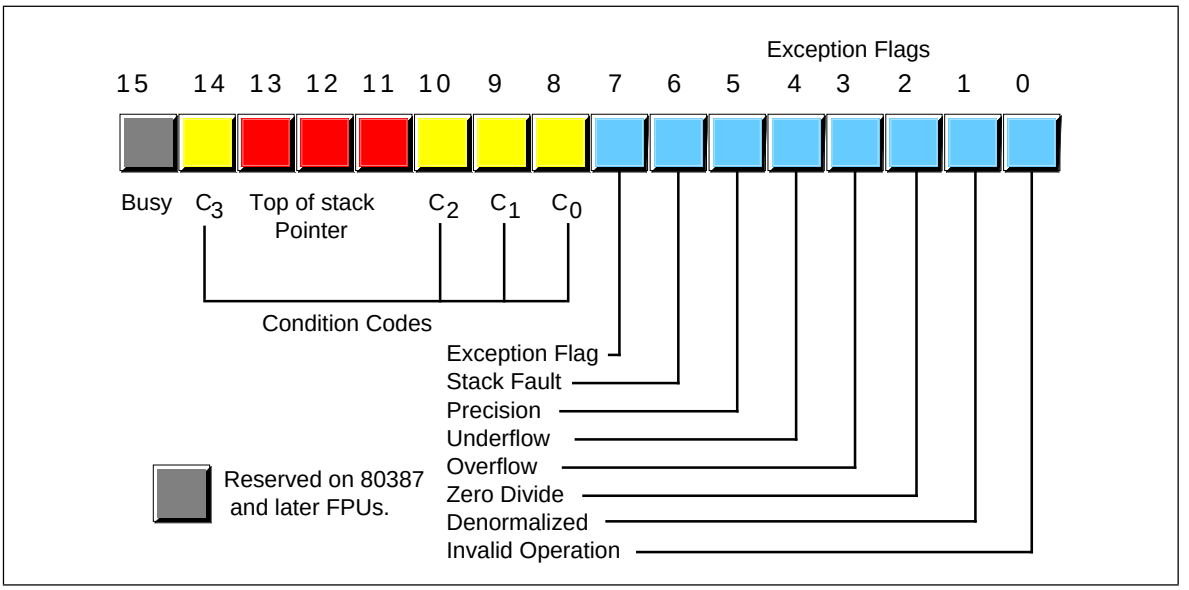


Figure 14.7 FPU Status Register

14.4.1.3 The FPU Status Register

The FPU status register provides the status of the coprocessor at the instant you read it. The FSTSW instruction stores the16 bit floating point status register into the mod/reg/rm operand. The status register s a 16 bit register, its layout appears in Figure 14.7.

Bits zero through five are the exception flags. These bits are appear in the same order as the exception masks in the control register. If the corresponding condition exists, then the bit is set. These bits are independent of the exception masks in the control register. The 80x87 sets and clears these bits regardless of the corresponding mask setting.

Bit six (active only on 80386 and later processors) indicates a *stack fault*. A stack fault occurs whenever there is a stack overflow or underflow. When this bit is set, the C_1 condition code bit determines whether there was a stack overflow (C_1=1) or stack underflow (C_1=0) condition.

Bit seven of the status register is set if *any* error condition bit is set. It is the logical OR of bits zero through five. A program can test this bit to quickly determine if an error condition exists.

Bits eight, nine, ten, and fourteen are the coprocessor condition code bits. Various instructions set the condition code bits as shown in the following table:

Table 60: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
fcom, fcomp, fcompp, ficom, ficomp	0	0	X	0	ST > source
	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	ST or source undefined
	X = Don't care				

Table 60: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
ftst	0	0	X	0	ST is positive
	0	0	X	1	ST is negative
	1	0	X	0	ST is zero (+ or -)
	1	1	X	1	ST is uncomparable
fxam	0	0	0	0	+ Unnormalized
	0	0	1	0	-Unnormalized
	0	1	0	0	+Normalized
	0	1	1	0	-Normalized
	1	0	0	0	+0
	1	0	1	0	-0
	1	1	0	0	+Denormalized
	1	1	1	0	-Denormalized
	0	0	0	1	+NaN
	0	0	1	1	-NaN
	0	1	0	1	+Infinity
	0	1	1	1	-Infinity
	1	X	X	1	Empty register
fucom, fucomp, fucompp	0	0	X	0	ST > source
	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	Unorder
	X = Don’t care				

Table 61: Condition Code Interpretation

Insruction(s)	C_0	C_3	C_2	C_1
fcom, fcomp, fcmpp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Result of comparison. See table above.	Result of comparison. See table above.	Operand is not comparable.	Result of comparison (see table above) or stack overflow / underflow (if stack exception bit is set).
fxam	See previous table.	See previous table.	See previous table.	Sign of result, or stack overflow / underflow (if stack exception bit is set).
fprem, fprem1	Bit 2 of remainder	Bit 0 of remainder	0- reduction done. 1- reduction incomplete.	Bit 1 of remainder or stack overflow / underflow (if stack exception bit is set).
fist, fbstp, frndint, fst, fstp, fadd, fmul, fdiv, fdivr, fsub, fsubr, fscale, fsqrt, fpatan, f2xm1, fyl2x, fyl2xp1	Undefined	Undefined	Undefined	Round up occurred or stack overflow / underflow (if stack exception bit is set).
fptan, fsin, fcos, fsincos	Undefined	Undefined	0- reduction done. 1- reduction incomplete.	Round up occurred or stack overflow / underflow (if stack exception bit is set).
fchs, fabs, fxch, fincstp, fdecstp, *constant loads* , fxtract, fld, fild, fbld, fstp (80 bit)	Undefined	Undefined	Undefined	Zero result or stack overflow / underflow (if stack exception bit is set).
fldenv, fstor	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.
fldcw, fstenv, fstcw, fstsw, fclex	Undefined	Undefined	Undefined	Undefined
finit, fsave	Cleared to zero.	Cleared to zero.	Cleared to zero.	Cleared to zero.

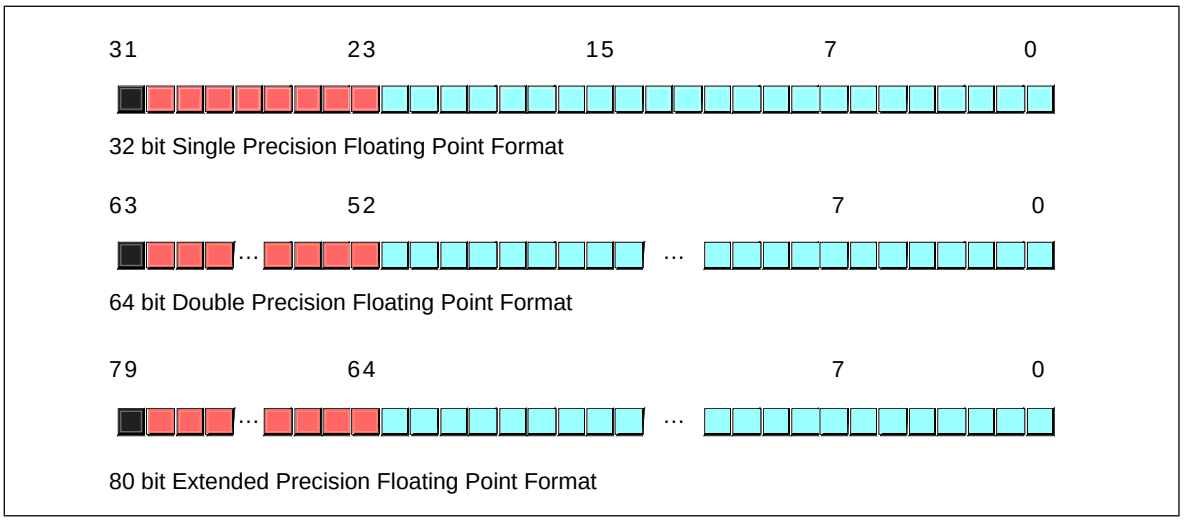


Figure 14.8 80x87 Floating Point Formats

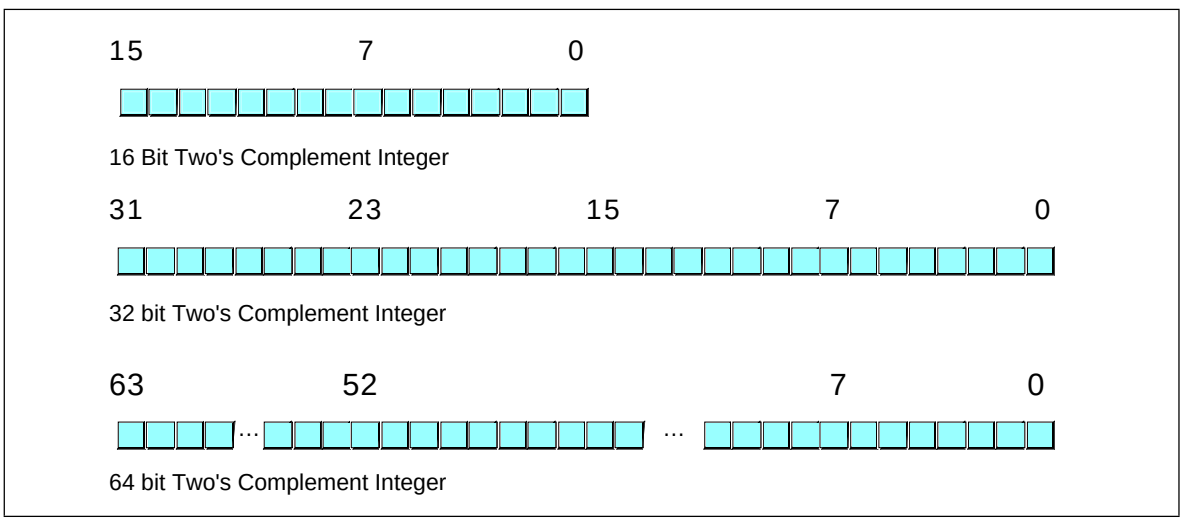


Figure 14.9 80x87 Integer Formats

Bits 11-13 of the FPU status register provide the register number of the top of stack. During computations, the 80x87 adds (modulo eight) the *logical* register numbers supplied by the programmer to these three bits to determine the *physical* register number at run time.

Bit 15 of the status register is the *busy* bit. It is set whenever the FPU is busy. Most programs will have little reason to access this bit.

14.4.2 FPU Data Types

The 80x87 FPU supports seven different data types: three integer types, a packed decimal type, and three floating point types. Since the 80x86 CPUs already support integer data types, these are few reasons why you would want to use the 80x87 integer types. The packed decimal type provides a 17 digit signed decimal (BCD) integer. However, we are avoiding BCD arithmetic in this text, so we will ignore this data type in the 80x87 FPU. The remaining three data types are the 32 bit, 64 bit, and 80 bit floating point data types we've looked at so far. The 80x87 data types appear in Figure 14.8, Figure 14.9, and Figure 14.10.

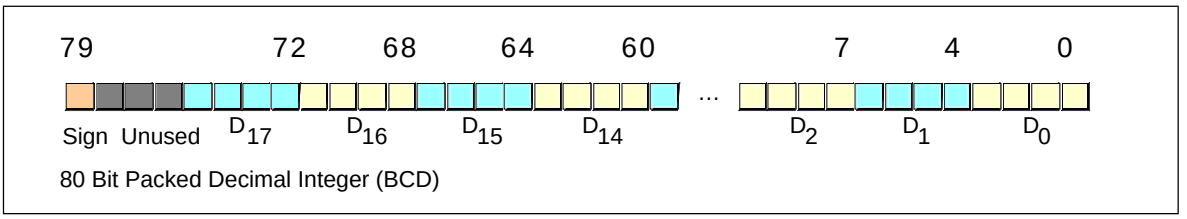


Figure 14.10 80x87 Packed Decimal Formats

The 80x87 FPU generally stores values in a *normalized* format. When a floating point number is normalized, the H.O. bit is always one. In the 32 and 64 bit floating point formats, the 80x87 does not actually store this bit, the 80x87 always assumes that it is one. Therefore, 32 and 64 bit floating point numbers are always normalized. In the extended precision 80 bit floating point format, the 80x87 does *not* assume that the H.O. bit of the mantissa is one, the H.O. bit of the number appears as part of the string of bits.

Normalized values provide the greatest precision for a given number of bits. However, there are a large number of non-normalized values which we *can* represent with the 80 bit format. These values are very close to zero and represent the set of values whose mantissa H.O. bit is not zero. The 80x87 FPUs support a special form of 80 bit known as *denormalized* values. Denormalized values allow the 80x87 to encode very small values it cannot encode using normalized values, but at a price. Denormalized values offer less bits of precision than normalized values. Therefore, using denormalized values in a computation may introduce some slight inaccuracy into a computation. Of course, this is always better than underflowing the denormalized value to zero (which could make the computation even less accurate), but you must keep in mind that if you work with very small values you may lose some accuracy in your computations. Note that the 80x87 status register contains a bit you can use to detect when the FPU uses a denormalized value in a computation.

14.4.3 The FPU Instruction Set

The 80387 (and later) FPU adds over 80 new instructions to the 80x86 instruction set. We can classify these instructions as *data movement instructions, conversions, arithmetic instructions, comparisons, constant instructions, transcendental instructions,* and *miscellaneous instructions.* The following sections describe each of the instructions in these categories.

14.4.4 FPU Data Movement Instructions

The data movement instructions transfer data between the internal FPU registers and memory. The instructions in this category are fld, fst, fstp, and fxch. The fld instructions always pushes its operand onto the floating point stack. The fstp instruction always pops the top of stack after storing the top of stack (tos) into its operation. The remaining instructions do not affect the number of items on the stack.

14.4.4.1 The FLD Instruction

The fld instruction loads a 32 bit, 64 bit, or 80 bit floating point value onto the stack. This instruction converts 32 and 64 bit operand to an 80 bit extended precision value before pushing the value onto the floating point stack.

The fld instruction first decrements the tos pointer (bits 11-13 of the status register) and then stores the 80 bit value in the physical register specified by the new tos pointer. If the source operand of the fld instruction is a floating point data register, ST(*i*), then the actual

register the 80x87 uses for the load operation is the register number *before* decrementing the tos pointer. Therefore, fld st or fld st(0) duplicates the value on the top of the stack.

The fld instruction sets the stack fault bit if stack overflow occurs. It sets the the denormalized exception bit if you load an 80 bit denormalized value. It sets the invalid operation bit if you attempt to load an empty floating point register onto the stop of stack (or perform some other invalid operation).

Examples:

```
fld     st(1)
fld     mem_32
fld     MyRealVar
fld     mem_64[bx]
```

14.4.4.2 The FST and FSTP Instructions

The fst and fstp instructions copy the value on the top of the floating point register stack to another floating point register or to a 32, 64, or 80 bit memory variable. When copying data to a 32 or 64 bit memory variable, the 80 bit extended precision value on the top of stack is rounded to the smaller format as specified by the rounding control bits in the FPU control register.

The fstp instruction pops the value off the top of stack when moving it to the destination location. It does this by incrementing the top of stack pointer in the status register after accessing the data in st(0). If the destination operand is a floating point register, the FPU stores the value at the specified register number *before* popping the data off the top of the stack.

Executing an fstp st(0) instruction effectively pops the data off the top of stack with no data transfer. Examples:

```
fst     mem_32
fstp    mem_64
fstp    mem_64[ebx*8]
fst     mem_80
fst     st(2)
fstp    st(1)
```

The last example above effectively pops st(1) while leaving st(0) on the top of the stack.

The fst and fstp instructions will set the stack exception bit if a stack underflow occurs (attempting to store a value from an empty register stack). They will set the precision bit if there is a loss of precision during the store operation (this will occur, for example, when storing an 80 bit extended precision value into a 32 or 64 bit memory variable and there are some bits lost during conversion). They will set the underflow exception bit when storing an 80 bit value value into a 32 or 64 bit memory variable, but the value is too small to fit into the destination operand. Likewise, these instructions will set the overflow exception bit if the value on the top of stack is too big to fit into a 32 or 64 bit memory variable. The fst and fstp instructions set the denormalized flag when you try to store a denormalized value into an 80 bit register or variable[7]. They set the invalid operation flag if an invalid operation (such as storing into an empty register) occurs. Finally, these instructions set the C_1 condition bit if rounding occurs during the store operation (this only occurs when storing into a 32 or 64 bit memory variable and you have to round the mantissa to fit into the destination).

14.4.4.3 The FXCH Instruction

The fxch instruction exchanges the value on the top of stack with one of the other FPU registers. This instruction takes two forms: one with a single FPU register as an operand,

7. Storing a denormalized value into a 32 or 64 bit memory variable will always set the underflow exception bit.

the second without any operands. The first form exchanges the top of stack with the specified register. The second form of fxch swaps the top of stack with st(1).

Many FPU instructions, e.g., fsqrt, operate only on the top of the register stack. If you want to perform such an operation on a value that is not on the top of stack, you can use the fxch instruction to swap that register with tos, perform the desired operation, and then use the fxch to swap the tos with the original register. The following example takes the square root of st(2):

```
fxch    st(2)
fsqrt
fxch    st(2)
```

The fxch instruction sets the stack exception bit if the stack is empty. It sets the invalid operation bit if you specify an empty register as the operand. This instruction always clears the C_1 condition code bit.

14.4.5 Conversions

The 80x87 chip performs all arithmetic operations on 80 bit real quantities. In a sense, the fld and fst/fstp instructions are conversion instructions as well as data movement instructions because they automatically convert between the internal 80 bit real format and the 32 and 64 bit memory formats. Nonetheless, we'll simply classify them as data movement operations, rather than conversions, because they are moving real values to and from memory. The 80x87 FPU provides five routines which convert to or from integer or binary coded decimal (BCD) format when moving data. These instructions are fild, fist, fistp, fbld, and fbstp.

14.4.5.1 The FILD Instruction

The fild (integer load) instruction converts a 16, 32, or 64 bit two's complement integer to the 80 bit extended precision format and pushes the result onto the stack. This instruction always expects a single operand. This operand must be the address of a word, double word, or quad word integer variable. Although the instruction format for fild uses the familiar mod/rm fields, the operand must be a memory variable, even for 16 and 32 bit integers. You cannot specify one of the 80386's 16 or 32 bit general purpose registers. If you want to push an 80x86 general purpose register onto the FPU stack, you must first store it into a memory variable and then use fild to push that value of that memory variable.

The fild instruction sets the stack exception bit and C_1 (accordingly) if stack overflow occurs while pushing the converted value. Examples:

```
fild    mem_16
fild    mem_32[ecx*4]
fild    mem_64[ebx+ecx*8]
```

14.4.5.2 The FIST and FISTP Instructions

The fist and fistp instructions convert the 80 bit extended precision variable on the top of stack to a 16, 32, or 64 bit integer and store the result away into the memory variable specified by the single operand. These instructions convert the value on tos to an integer according to the rounding setting in the FPU control register (bits 10 and 11). As for the fild instruction, the fist and fistp instructions will not let you specify one of the 80x86's general purpose 16 or 32 bit registers as the destination operand.

The fist instruction converts the value on the top of stack to an integer and then stores the result; it does not otherwise affect the floating point register stack. The fistp instruction pops the value off the floating point register stack after storing the converted value.

These instructions set the stack exception bit if the floating point register stack is empty (this will also clear C_1). They set the precision (imprecise operation) and C_1 bits if rounding occurs (that is, if there is any fractional component to the value in st(0)). These instructions set the underflow exception bit if the result is too small (i.e., less than one but greater than zero or less than zero but greater than -1). Examples:

```
fist    mem_16[bx]
fist    mem_64
fistp   mem_32
```

Don't forget that these instructions use the rounding control settings to determine how they will convert the floating point data to an integer during the store operation. Be default, the rouding control is usually set to "round" mode; yet most programmers expect fist/fistp to truncate the decimal portion during conversion. If you want fist/fistp to truncate floating point values when converting them to an integer, you will need to set the rounding control bits appropriately in the floating point control register.

14.4.5.3 The FBLD and FBSTP Instructions

The fbld and fbstp instructions load and store 80 bit BCD values. The fbld instruction converts a BCD value to its 80 bit extended precision equivalent and pushes the result onto the stack. The fbstp instruction pops the extended precision real value on tos, converts it to an 80 bit BCD value (rounding according to the bits in the floating point control register), and stores the converted result at the address specified by the destination memory operand. Note that there is no fbst instruction which stores the value on tos without popping it.

The fbld instruction sets the stack exception bit and C_1 if stack overflow occurs. It sets the invalid operation bit if you attempt to load an invalid BCD value. The fbstp instruction sets the stack exception bit and clears C_1 if stack underflow occurs (the stack is empty). It sets the underflow flag under the same conditions as fist and fistp. Examples:

```
; Assuming fewer than eight items on the stack, the following
; code sequence is equivalent to an fbst instruction:

                fld     st(0)       ;Duplicate value on TOS.
                fbstp   mem_80

; The following example easily converts an 80 bit BCD value to
; a 64 bit integer:

                fbld    bcd_80      ;Get BCD value to convert.
                fist    mem_64      ;Store as an integer.
```

14.4.6 Arithmetic Instructions

The arithmetic instructions make up a small, but important, subset of the 80x87's instruction set. These instructions fall into two general categories – those which operate on real values and those which operate on a real and an integer value.

14.4.6.1 The FADD and FADDP Instructions

These two instructions take the following forms:

```
fadd
faddp
fadd    st(i), st(0)
fadd    st(0), st(i)
faddp   st(i), st(0)
fadd    mem
```

The first two forms are equivalent. They pop the two values on the top of stack, add them, and push their sum back onto the stack.

The next two forms of the fadd instruction, those with two FPU register operands, behave like the 80x86's add instruction. They add the value in the second register operand to the value in the first register operand. Note that one of the register operands must be st(0)[8].

The faddp instruction with two operands adds st(0) (which must always be the second operand) to the destination (first) operand and then pops st(0). The destination operand must be one of the other FPU registers.

The last form above, fadd with a memory operand, adds a 32 or 64 bit floating point variable to the value in st(0). This instruction will convert the 32 or 64 bit operands to an 80 bit extended precision value before performing the addition. Note that this instruction does *not* allow an 80 bit memory operand.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

14.4.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions

These four instructions take the following forms:

```
fsub
fsubp
fsubr
fsubrp

fsub     st(i). st(0)
fsub     st(0), st(i)
fsubp    st(i), st(0)
fsub     mem

fsubr    st(i). st(0)
fsubr    st(0), st(i)
fsubrp   st(i), st(0)
fsubr    mem
```

With no operands, the fsub and fsubp instructions operate identically. They pop st(0) and st(1) from the register stack, compute st(0)-st(1), and the push the difference back onto the stack. The fsubr and fsubrp instructions (reverse subtraction) operate in an almost identical fashion except they compute st(1)-st(0) and push that difference.

With two register operands (*destination, source*) the fsub instruction computes *destination := destination - source*. One of the two registers must be st(0). With two registers as operands, the fsubp also computes *destination := destination - source* and then it pops st(0) off the stack after computing the difference. For the fsubp instruction, the source operand must be st(0).

With two register operands, the fsubr and fsubrp instruction work in a similar fashion to fsub and fsubp, except they compute *destination := source - destination.*

The fsub mem and fsubr mem instructions accept a 32 or 64 bit memory operand. They convert the memory operand to an 80 bit extended precision value and subtract this from st(0) (fsub) or subtract st(0) from this value (fsubr) and store the result back into st(0).

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

8. Because you will use st(0) quite a bit when programming the 80x87, MASM allows you to use the abbreviation st for st(0). However, this text will explicitly state st(0) so there will be no confusion.

14.4.6.3 The FMUL and FMULP Instructions

The fmul and fmulp instructions multiply two floating point values. These instructions allow the following forms:

```
fmul
fmulp

fmul     st(0), st(i)
fmul     st(i), st(0)
fmul     mem

fmulp    st(i), st(0)
```

With no operands, fmul and fmulp both do the same thing – they pop st(0) and st(1), multiply these values, and push their product back onto the stack. The fmul instructions with two register operands compute *destination := destination * source*. One of the registers (source or destination) must be st(0).

The fmulp st(i), st(0) instruction computes st(i) := st(i) * st(0) and then pops st(0). This instruction uses the value for *i* before popping st(0). The fmul mem instruction requires a 32 or 64 bit memory operand. It converts the specified memory variable to an 80 bit extended precision value and the multiplies st(0) by this value.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

14.4.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions

These four instructions allow the following forms:

```
fdiv
fdivp
fdivr
fdivrp

fdiv     st(0), st(i)
fdiv     st(i), st(0)
fdivp    st(i), st(0)

fdivr    st(0), st(i)
fdivr    st(i), st(0)
fdivrp   st(i), st(0)

fdiv     mem
fdivr    mem
```

With zero operands, the fdiv and fdivp instructions pop st(0) and st(1), compute st(0)/st(1), and push the result back onto the stack. The fdivr and fdivrp instructions also pop st(0) and st(1) but compute st(1)/st(0) before pushing the quotient onto the stack.

With two register operands, these instructions compute the following quotients:

```
fdiv     st(0), st(i)          ;st(0) := st(0)/st(i)
fdiv     st(i), st(0)          ;st(i) := st(i)/st(0)
fdivp    st(i), st(0)          ;st(i) := st(i)/st(0)
fdivr    st(i), st(i)          ;st(0) := st(0)/st(i)
fdivrp   st(i), st(0)          ;st(i) := st(0)/st(i)
```

The fdivp and fdivrp instructions also pop st(0) after performing the division operation. The value for *i* in this two instructions is computed before popping st(0).

These instructions can raise the stack, precision, underflow, overflow, denormalized, zero divide, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

14.4.6.5 The FSQRT Instruction

The fsqrt routine does not allow any operands. It computes the square root of the value on tos and replaces st(0) with this result. The value on tos must be zero or positive, otherwise fsqrt will generate an invalid operation exception.

This instruction can raise the stack, precision, denormalized, and invalid operation exceptions, as appropriate. If rounding occurs during the computation, fsqrt sets the C_1 condition code bit. If a stack fault exception occurs, C_1 denotes stack overflow or underflow.

Example:

```
; Compute Z := sqrt(x**2 + y**2);

                fld     x           ;Load X.
                fld     st(0)       ;Duplicate X on TOS.
                fmul                ;Compute X**2.

                fld     y           ;Load Y.
                fld     st(0)       ;Duplicate Y on TOS.
                fmul                ;Compute Y**2.

                fadd                ;Compute X**2 + Y**2.
                fsqrt               ;Compute sqrt(x**2 + y**2).
                fst     Z           ;Store away result in Z.
```

14.4.6.6 The FSCALE Instruction

The fscale instruction pops two values off the stack. It multiplies st(0) by $2^{st(1)}$ and pushes the result back onto the stack. If the value in st(1) is not an integer, fscale truncates it towards zero before performing the operation.

This instruction raises the stack exception if there are not two items currently on the stack (this will also clear C_1 since stack underflow occurs). It raises the precision exception if there is a loss of precision due to this operation (this occurs when st(1) contains a large, negative, value). Likewise, this instruction sets the underflow or overflow exception bits if you multiply st(0) by a very large positive or negative power of two. If the result of the multiplication is very small, fscale could set the denormalized bit. Also, this instruction could set the invalid operation bit if you attempt to fscale illegal values. Fscale sets C_1 if rounding occurs in an otherwise correct computation. Example:

```
                fild    Sixteen     ;Push sixteen onto the stack.
                fld     x           ;Compute x * (2**16).
                fscale
                  .
                  .
                  .
Sixteen         word    16
```

14.4.6.7 The FPREM and FPREM1 Instructions

The fprem and fprem1 instructions compute a *partial remainder*. Intel designed the fprem instruction before the IEEE finalized their floating point standard. In the final draft of the IEEE floating point standard, the definition of fprem was a little different than Intel's original design. Unfortunately, Intel needed to maintain compatibility with the existing software that used the fprem instruction, so they designed a new version to handle the IEEE partial remainder operation, fprem1. You should always use fprem1 in new software you write, therefore we will only discuss fprem1 here, although you use fprem in an identical fashion.

Fprem1 computes the *partial* remainder of st(0)/st(1). If the difference between the exponents of st(0) and st(1) is less than 64, fprem1 can compute the exact remainder in one

operation. Otherwise you will have to execute the fprem1 two or more times to get the correct remainder value. The C_2 condition code bit determines when the computation is complete. Note that fprem1 does *not* pop the two operands off the stack; it leaves the partial remainder in st(0) and the original divisor in st(1) in case you need to compute another partial product to complete the result.

The fprem1 instruction sets the stack exception flag if there aren't two values on the top of stack. It sets the underflow and denormal exception bits if the result is too small. It sets the invalid operation bit if the values on tos are inappropriate for this operation. It sets the C_2 condition code bit if the partial remainder operation is not complete. Finally, it loads C_3, C_1, and C_0 with bits zero, one, and two of the quotient, respectively.

Example:

```
; Compute Z := X mod Y

                fld     y
                fld     x
PartialLp:      fprem1
                fstsw   ax          ;Get condition bits in AX.
                test    ah, 100b    ;See if C2 is set.
                jnz     PartialLp   ;Repeat if not done yet.
                fstp    Z           ;Store remainder away.
                fstp    st(0)       ;Pop old y value.
```

14.4.6.8 The FRNDINT Instruction

The frndint instruction rounds the value on tos to the nearest integer using the rounding algorithm specified in the control register.

This instruction sets the stack exception flag if there is no value on the tos (it will also clear C_1 in this case). It sets the precision and denormal exception bits if there was a loss of precision. It sets the invalid operation flag if the value on the tos is not a valid number.

14.4.6.9 The FXTRACT Instruction

The fxtract instruction is the complement to the fscale instruction. It pops the value off the top of the stack and pushes a value which is the integer equivalent of the exponent (in 80 bit real form), and then pushes the mantissa with an exponent of zero (3fffh in biased form).

This instruction raises the stack exception if there is a stack underflow when popping the original value or a stack overflow when pushing the two results (C_1 determines whether stack overflow or underflow occurs). If the original top of stack was zero, fxtract sets the zero division exception flag. The denormalized flag is set if the result warrants it; and the invalid operation flag is set if there are illegal input values when you execute fxtract.

Example:

```
; The following example extracts the binary exponent of X and
; stores this into the 16 bit integer variable Xponent.

                fld     x
                fxtract
                fstp    st(0)
                fistp   Xponent
```

14.4.6.10 The FABS Instruction

Fabs computes the absolute value of st(0) by clearing the sign bit of st(0). It sets the stack exception bit and invalid operation bits if the stack is empty.

Example:

```
; Compute X := sqrt(abs(x));

                fld     x
                fabs
                fsqrt
                fstp    x
```

14.4.6.11 The FCHS Instruction

Fchs changes the sign of st(0)'s value by inverting its sign bit. It sets the stack exception bit and invalid operation bits if the stack is empty. Example:

```
; Compute X := -X if X is positive, X := X if X is negative.

                fld     x
                fabs
                fchs
                fstp    x
```

14.4.7 Comparison Instructions

The 80x87 provides several instructions for comparing real values. The fcom, fcomp, fcompp, fucom, fucomp, and fucompp instructions compare the two values on the top of stack and set the condition codes appropriately. The ftst instruction compares the value on the top of stack with zero. The fxam instrution checks the value on tos and reports sign, normalization, and tag information.

Generally, most programs test the condition code bits immediately after a comparison. Unfortunately, there are no conditional jump instructions that branch based on the FPU condition codes. Instead, you can use the fstsw instruction to copy the floating point status register (see "The FPU Status Register" on page 785) into the ax register; then you can use the sahf instruction to copy the ah register into the 80x86's condition code bits. After doing this, you can can use the conditional jump instructions to test some condition. This technique copies C_0 into the carry flag, C_2 into the parity flag, and C_3 into the zero flag. The sahf instruction does not copy C_1 into any of the 80x86's flag bits.

Since the sahf instruction does not copy any 80x87 processor status bits into the sign or overflow flags, you cannot use the jg, jl, jge, or jle instructions. Instead, use the ja, jae, jb, jbe, je, and jz instructions when testing the results of a floating point comparison. *Yes, these conditional jumps normally test unsigned values and floating point numbers are signed values.* However, use the unsigned conditional branches anyway; the fstsw and sahf instructions set the 80x86 flags register to use the unsigned jumps.

14.4.7.1 The FCOM, FCOMP, and FCOMPP Instructions

The fcom, fcomp, and fcompp instructions compare st(0) to the specified operand and set the corresponding 80x87 condition code bits based on the result of the comparison. The legal forms for these instructions are

```
fcom
fcomp
fcompp

fcom    st(i)
fcomp   st(i)

fcom    mem
fcomp   mem
```

With no operands, fcom, fcomp, and fcompp compare st(0) against st(1) and set the processor flags accordingly. In addition, fcomp pops st(0) off the stack and fcompp pops both st(0) and st(1) off the stack.

With a single register operand, fcom and fcomp compare st(0) against the specified register. Fcomp also pops st(0) after the comparison.

With a 32 or 64 bit memory operand, the fcom and fcomp instructions convert the memory variable to an 80 bit extended precision value and then compare st(0) against this value, setting the condition code bits accordingly. Fcomp also pops st(0) after the comparison.

These instructions set C_2 (which winds up in the parity flag) if the two operands are not comparable (e.g., NaN). If it is possible for an illegal floating point value to wind up in a comparison, you should check the parity flag for an error before checking the desired condition.

These instructions set the stack fault bit if there aren't two items on the top of the register stack. They set the denormalized exception bit if either or both operands are denormalized. They set the invalid operation flag if either or both operands are quite NaNs. These instructions always clear the C_1 condition code.

14.4.7.2 The FUCOM, FUCOMP, and FUCOMPP Instructions

These instructions are similar to the fcom, fcomp, and fcompp instructions, although they only allow the following forms:

```
fucom
fucomp
fucompp
fucom    st(i)
fucomp   st(i)
```

The difference between fcom/fcomp/fcompp and fucom/fucomp/fucompp is relatively minor. The fcom/fcomp/fcompp instructions set the invalid operation exception bit if you compare two NaNs. The fucom/fucomp/fucompp instructions do not. In all other cases, these two sets of instructions behave identically.

14.4.7.3 The FTST Instruction

The ftst instruction compares the value in st(0) against 0.0. It behaves just like the fcom instruction would if st(1) contained 0.0. Note that this instruction does not differentiate -0.0 from +0.0. If the value in st(0) is either of these values, ftst will set C_3 to denote equality. If you need to differentiate -0.0 from +0.0, use the fxam instruction. Note that this instruction does *not* pop st(0) off the stack.

14.4.7.4 The FXAM Instruction

The fxam instruction examines the value in st(0) and reports the results in the condition code bits (see "The FPU Status Register" on page 785 for details on how fxam sets these bits). This instruction does not pop st(0) off the stack.

14.4.8 Constant Instructions

The 80x87 FPU provides several instructions that let you load commonly used constants onto the FPU's register stack. These instructions set the stack fault, invalid opera-

tion, and C_1 flags if a stack overflow occurs; they do not otherwise affect the FPU flags. The specific instructions in this category include:

```
fldz                ;Pushes +0.0.
fld1                ;Pushes +1.0.
fldpi               ;Pushes π.
fldl2t              ;Pushes log2(10).
fldl2e              ;Pushes log2(e).
fldlg2              ;Pushes log10(2).
fldln2              ;Pushes ln(2).
```

14.4.9 Transcendental Instructions

The 80387 and later FPUs provide eight transcendental (log and trigonometric) instructions to compute a partial tangent, partial arctangent, 2^x-1, y * $\log_2(x)$, and y * $\log_2(x+1)$. Using various algebraic identities, it is easy to compute most of the other common transcendental functions using these instructions.

14.4.9.1 The F2XM1 Instruction

F2xm1 computes $2^{st(0)}$-1. The value in st(0) must be in the range -1.0 ≤ st(0) ≤ +1.0. If st(0) is out of range f2xm1 generates an undefined result but raises no exceptions. The computed value replaces the value in st(0). Example:

```
; Compute 10^x using the identity: 10^x = 2^(x*lg(10)) (lg = log2).

                fld     x
                fldl2t
                fmul
                f2xm1
                fld1
                fadd
```

Note that f2xm1 computes 2^x-1, which is why the code above adds 1.0 to the result at the end of the computation.

14.4.9.2 The FSIN, FCOS, and FSINCOS Instructions

These instructions pop the value off the top of the register stack and compute the sine, cosine, or both, and push the result(s) back onto the stack. The fsincos pushes the sine followed by the cosine of the original operand, hence it leaves cos(st(0)) in st(0) and sin(st(0)) in st(1).

These instructions assume st(0) specifies an angle in radians and this angle must be in the range -2^{63} < st(0) < $+2^{63}$. If the original operand is out of range, these instructions set the C_2 flag and leave st(0) unchanged. You can use the fprem1 instruction, with a divisor of 2π, to reduce the operand to a reasonable range.

These instructions set the stack fault/C_1, precision, underflow, denormalized, and invalid operation flags according to the result of the computation.

14.4.9.3 The FPTAN Instruction

Fptan computes the tangent of st(0) and pushes this value and then it pushes 1.0 onto the stack. Like the fsin and fcos instructions, the value of st(0) is assumed to be in radians and must be in the range -2^{63}<st(0)<$+2^{63}$. If the value is outside this range, fptan sets C_2 to indicate that the conversion did not take place. As with the fsin, fcos, and fsincos instructions, you can use the fprem1 instruction to reduce this operand to a reasonable range using a divisor of 2π.

If the argument is invalid (i.e., zero or π radians, which causes a division by zero) the result is undefined and this instruction raises no exceptions. Fptan will set the stack fault, precision, underflow, denormal, invalid operation, C_2, and C_1 bits as required by the operation.

14.4.9.4 The FPATAN Instruction

This instruction expects two values on the top of stack. It pops them and computes the following:

$st(0) = \tan^{-1}(st(1) / st(0))$

The resulting value is the arctangent of the ratio on the stack expressed in radians. If you have a value you wish to compute the tangent of, use fld1 to create the appropriate ratio and then execute the fpatan instruction.

This instruction affects the stack fault/C_1, precision, underflow, denormal, and invalid operation bits if an problem occurs during the computation. It sets the C_1 condition code bit if it has to round the result.

14.4.9.5 The FYL2X and FYL2XP1 Instructions

The fyl2x and fyl2xp1 instructions compute st(1) * $\log_2$(st(0)) and st(1) * $\log_2$(st(0)+1), respectively. Fyl2x requires that st(0) be greater than zero, fyl2xp1 requires st(0) to be in the range:

$$\left(-1-\left(\frac{\sqrt{2}}{2}\right)\right) < st(0) < \left(1-\left(\frac{\sqrt{2}}{2}\right)\right)$$

Fyl2x is useful for computing logs to bases other than two; fyl2xp1 is useful for computing compound interest, maintaining the maximum precision during computation.

Fyl2x can affect *all* the exception flags. C_1 denotes rounding if there is not other error, stack overflow / underflow if the stack fault bit is set.

The fyl2xp1 instruction does not affect the overflow or zero divide exception flags. These exceptions occur when st(0) is very small or zero. Since fyl2xp1 adds one to st(0) before computing the function, this condition never holds. Fyl2xp1 affects the other flags in a manner identical to fyl2x.

14.4.10 Miscellaneous instructions

The 80x87 FPU includes several additional instructions which control the FPU, synchronize operations, and let you test or set various status bits. These instructions include finit/fninit, fdisi/fndisi, feni/fneni, fldcw, fstcw/fnstcw, fclex/fnclex, fsave/fnsave, frstor, frstpm, fstsw/fnstsw, fstenv/fnstenv, fldenv, fincstp, fdecstp, fwait, fnop, and ffree. The fdisi/fndisi, feni/fneni, and frstpm are active only on FPUs earlier than the 80387, so we will not consider them here.

Many of these instructions have two forms. The first form is Fxxxx and the second form is FNxxxx. The version without the "N" emits an fwait instruction prior to opcode (which is standard for most coprocessor instructions). The version with the "N" does not emit the fwait opcode ("N" stands for *no wait*).

14.4.10.1 The FINIT and FNINIT Instructions

The finit instruction intializes the FPU for proper operation. Your applications should execute this instruction before executing any other FPU instructions. This instruction ini-

tializes the control register to 37Fh (see “The FPU Control Register” on page 782), the status register to zero (see “The FPU Status Register” on page 785) and the tag word to 0FFFFh. The other registers are unaffected.

14.4.10.2 The FWAIT Instruction

The fwait instruction pauses the system until any currently executing FPU instruction completes. This is required because the FPU on the 80486sx and earlier CPU/FPU combinations can execute instructions in parallel with the CPU. Therefore, any FPU instruction which reads or writes memory could suffer from a data hazard if the main CPU accesses that same memory location before the FPU reads or writes that location. The fwait instruction lets you synchronize the operation of the FPU by waiting until the completion of the current FPU instruction. This resolves the data hazard by, effectively, inserting an explict “stall” into the execution stream.

14.4.10.3 The FLDCW and FSTCW Instructions

The fldcw and fstcw instructions require a single 16 bit memory operand:

```
fldcw     mem_16
fstcw     mem_16
```

These two instructions load the control register (see “The FPU Control Register” on page 782) from a memory location (fldcw) or store the control word to a 16 bit memory location (fstcw).

When using the fldcw instruction to turn on one of the exceptions, if the corresponding exception flag is set when you enable that exception, the FPU will generate an immediate interrupt before the CPU executes the next instruction. Therefore, you should use the fclex instruction to clear any pending interrupts before changing the FPU exception enable bits.

14.4.10.4 The FCLEX and FNCLEX Instructions

The fclex and fnclex instructions clear all exception bits the stack fault bit, and the busy flag in the FPU status register (see “The FPU Status Register” on page 785).

14.4.10.5 The FLDENV, FSTENV, and FNSTENV Instructions

```
fstenv    mem_14b
fnstenv   mem_14b
fldenv    mem_14b
```

The fstenv/fnstenv instructions store a 14-byte FPU environment record to the memory operand specified. When operating in real mode (the only mode this text considers), the environment record takes the form appearing in Figure 14.11.

You must execute the fstenv and fnstenv instructions with the CPU interrupts disabled. Furthermore, you should always ensure that the FPU is not busy before executing this instruction. This is easily accomplished by using the following code:

```
pushf                ;Preserve I flag.
cli                  ;Disable interrupts.
fstenv    mem_14b    ;Implicit wait for not busy.
fwait                ;Wait for operation to finish.
popf                 ;Restore I flag.
```

The fldenv instruction loads the FPU environment from the specified memory operand. Note that this instruction lets you load the the status word. There is no explicit instruction like fldcw to accomplish this.

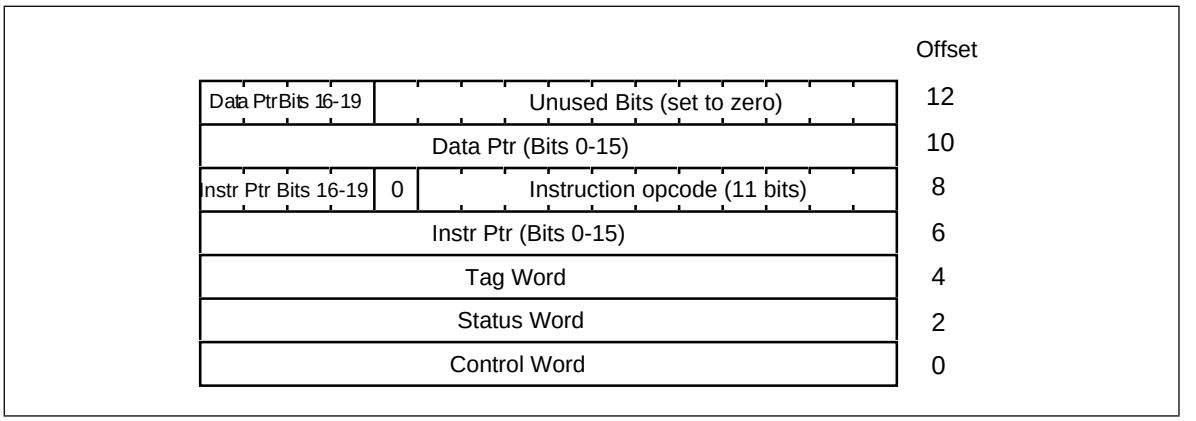


Figure 14.11 FPU Environment Record (16 Bit Real Mode)

14.4.10.6 The FSAVE, FNSAVE, and FRSTOR Instructions

```
fsave     mem_94b
fnsave    mem_94b
frstor    mem_94b
```

These instructions save and restore the state of the FPU. This includes saving all the internal control, status, and data registers. The destination location for fsave/fnsave (source location for frstor) must be 94 bytes long. The first 14 bytes correspond to the environment record the fldenv and fstenv instructions use; the remaining 80 bytes hold the data from the FPU register stack written out as st(0) through st(7). Frstor reloads the environment record and floating point registers from the specified memory operand.

The fsave/fnsave and frstor instructions are mainly intended for task switching. You can also use fsave/fnsave and frstor as a "push all" and "pop all" sequence to preserve the state of the FPU.

Like the fstenv and fldenv instructions, interrupts should be disabled while saving or restoring the FPU state. Otherwise another interrupt service routine could manipulate the FPU registers and invalidate the operation of the fsave/fnsave or frestore operation. The following code properly protects the environment data while saving and restore the FPU status:

```
; Preserve the FPU state, assume di points at the environment
; record in memory.

                pushf
                cli
                fsave     [si]
                fwait
                popf
                  .
                  .
                  .
                pushf
                cli
                frstor    [si]
                fwait
                popf
```

14.4.10.7 The FSTSW and FNSTSW Instructions

```
fstsw    ax
fnstsw   ax
fstsw    mem_16
fnstsw   mem_16
```

These instructions store the FPU status register (see “The FPU Status Register” on page 785) into a 16 bit memory location or the ax register. These instructions are unusual in the sense that they can copy an FPU value into one of the 80x86 general purpose registers. Of course, the whole purpose behind allowing the transfer of the status register into ax is to allow the CPU to easily test the condition code register with the sahf instruction.

14.4.10.8 The FINCSTP and FDECSTP Instructions

The fincstp and fdecstp instructions do not take any operands. They simply increment and decrement the stack pointer bits (mod 8) in the FPU status register. These two instructions clear the C_1 flag, but do not otherwise affect the condition code bits in the FPU status register.

14.4.10.9 The FNOP Instruction

The fnop instruction is simply an alias for fst st, st(0). It performs no other operation on the FPU.

14.4.10.10 The FFREE Instruction

```
ffree    st(i)
```

This instruction modifies the tag bits for register *i* in the tags register to mark the specified register as emtpy. The value is unaffected by this instruction, but the FPU will no longer be able to access that data (without resetting the appropriate tag bits).

14.4.11 Integer Operations

The 80x87 FPUs provide special instructions that combine integer to extended precision conversion along with various arithmetic and comparison operations. These instructions are the following:

```
fiadd    int
fisub    int
fisubr   int
fimul    int
fidiv    int
fidivr   int

ficom    int
ficomp   int
```

These instructions convert their 16 or 32 bit integer operands to an 80 bit extended precision floating point value and then use this value as the source operand for the specified operation. These instructions use st(0) as the destination operand.

14.5 Sample Program: Additional Trigonometric Functions

This section provides various examples of 80x87 FPU programming. This group of routines provides several trigonometric, inverse trigonometric, logarithmic, and exponential functions using various algebraic identities. All these functions assume that the input values are on the stack are are within valid ranges for the given functions. The trigonometric routines expect angles expressed in radians and the inverse trig routines produce angles measured in radians.

This program (transcnd.asm) appears on the companion CD-ROM.

```
                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

                .386
                .387
                option    segment:use16

dseg            segment   para public 'data'

result          real8     ?

; Some variables we use to test the routines in this package:

cotvar          real8     3.0
cotRes          real8     ?
acotRes         real8     ?

cscvar          real8     1.5
cscRes          real8     ?
acscRes         real8     ?

secvar          real8     0.5
secRes          real8     ?
asecRes         real8     ?

sinvar          real8     0.75
sinRes          real8     ?
asinRes         real8     ?

cosvar          real8     0.25
cosRes          real8     ?
acosRes         real8     ?

Two2xvar        real8     -2.5
Two2xRes        real8     ?
lgxRes          real8     ?

Ten2xVar        real8     3.75
Ten2xRes        real8     ?
logRes          real8     ?

expVar          real8     3.25
expRes          real8     ?
lnRes           real8     ?

Y2Xx            real8     3.0
Y2Xy            real8     3.0
Y2XRes          real8     ?

dseg            ends

cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg
```

```
; COT(x) - Computes the cotangent of st(0) and leaves result in st(0).
;                st(0) contains x (in radians) and must be between
;                -2**63 and +2**63
;
;       There must be at least one free register on the stack for
;       this routine to operate properly.
;
;      cot(x) = 1/tan(x)

cot             proc     near
                fsincos
                fdivr
                ret
cot             endp

; CSC(x) - computes the cosecant of st(0) and leaves result in st(0).
;                st(0) contains x (in radians) and must be between
;                -2**63 and +2**63.
;       The cosecant of x is undefined for any value of sin(x) that
;                produces zero (e.g., zero or pi radians).
;
;       There must be at least one free register on the stack for
;       this routine to operate properly.
;
;      csc(x) = 1/sin(x)

csc             proc     near
                fsin
                fld1
                fdivr
                ret
csc             endp

; SEC(x) - computes the secant of st(0) and leaves result in st(0).
;                st(0) contains x (in radians) and must be between
;                -2**63 and +2**63.
;
;      The secant of x is undefined for any value of cos(x) that
;      produces zero (e.g., pi/2 radians).
;
;      There must be at least one free register on the stack for
;      this routine to operate properly.
;
;      sec(x) = 1/cos(x)

sec             proc     near
                fcos
                fld1
                fdivr
                ret
sec             endp

; ASIN(x)- Computes the arcsine of st(0) and leaves the result in st(0).
;                Allowable range: -1<=x<=+1
;                There must be at least two free registers for this
;                function to operate properly.
;
;      asin(x) = atan(sqrt(x*x/(1-x*x)))

asin            proc     near
                fld      st(0)        ;Duplicate X on tos.
                fmul                  ;Compute X**2.
                fld      st(0)        ;Duplicate X**2 on tos.
                fld1                  ;Compute 1-X**2.
                fsubr
                fdiv                  ;Compute X**2/(1-X**2).
                fsqrt                 ;Compute sqrt(x**2/(1-X**2)).
                fld1                  ;To compute full arctangent.
                fpatan                ;Compute atan of the above.
                ret
```

```
asin            endp

; ACOS(x)- Computes the arccosine of st(0) and leaves the
;               result in st(0).
;               Allowable range: -1<=x<=+1
;               There must be at least two free registers for
;               this function to operate properly.
;
;       acos(x) = atan(sqrt((1-x*x)/(x*x)))

acos            proc    near
                fld     st(0)       ;Duplicate X on tos.
                fmul                ;Compute X**2.
                fld     st(0)       ;Duplicate X**2 on tos.
                fld1                ;Compute 1-X**2.
                fsubr
                fdiv                ;Compute (1-x**2)/X**2.
                fsqrt               ;Compute sqrt((1-X**2)/X**2).
                fld1                ;To compute full arctangent.
                fpatan              ;Compute atan of the above.
                ret
acos            endp

; ACOT(x)- Computes the arccotangent of st(0) and leaves the
;               result in st(0).
;               X cannot equal zero.
;               There must be at least one free register for
;               this function to operate properly.
;
;       acot(x) = atan(1/x)

acot            proc    near
                fld1                ;fpatan computes
                fxch                ; atan(st(1)/st(0)).
                fpatan              ; we want atan(st(0)/st(1)).
                ret
acot            endp

; ACSC(x)- Computes the arccosecant of st(0) and leaves the
;               result in st(0).
;               abs(X) must be greater than one.
;               There must be at least two free registers for
;               this function to operate properly.
;
;       acsc(x) = atan(sqrt(1/(x*x-1)))

acsc            proc    near
                fld     st(0)       ;Compute x*x
                fmul
                fld1                ;Compute x*x-1
                fsub
                fld1                ;Compute 1/(x*x-1)
                fdivr
                fsqrt               ;Compute sqrt(1/(x*x-1))
                fld1
                fpatan              ;Compute atan of above.
                ret
acsc            endp

; ASEC(x)- Computes the arcsecant of st(0) and leaves the
;               result in st(0).
;               abs(X) must be greater than one.
;               There must be at least two free registers for
;               this function to operate properly.
;
;       asec(x) = atan(sqrt(x*x-1))

asec            proc    near
                fld     st(0)       ;Compute x*x
                fmul
```

```
                fld1                    ;Compute x*x-1
                fsub
                fsqrt                   ;Compute sqrt(x*x-1)
                fld1
                fpatan                  ;Compute atan of above.
                ret
asec            endp

; TwoToX(x)- Computes 2**x.
;               It does this by using the algebraic identity:
;
;               2**x = 2**int(x) * 2**frac(x).
;               We can easily compute 2**int(x) with fscale and
;               2**frac(x) using f2xm1.
;
;               This routine requires three free registers.

SaveCW          word    ?
MaskedCW        word    ?

TwoToX          proc    near
                fstcw   cseg:SaveCW

; Modify the control word to truncate when rounding.

                fstcw   cseg:MaskedCW
                or      byte ptr cseg:MaskedCW+1, 1100b
                fldcw   cseg:MaskedCW

                fld     st(0)           ;Duplicate tos.
                fld     st(0)
                frndint                 ;Compute integer portion.

                fxch                    ;Swap whole and int values.
                fsub    st(0), st(1) ;Compute fractional part.

                f2xm1                   ;Compute 2**frac(x)-1.
                fld1
                fadd                    ;Compute 2**frac(x).

                fxch                    ;Get integer portion.
                fld1                    ;Compute 1*2**int(x).
                fscale
                fstp    st(1)           ;Remove st(1) (which is 1).

                fmul                    ;Compute 2**int(x) * 2**frac(x).

                fldcw   cseg:SaveCW  ;Restore rounding mode.
                ret
TwoToX          endp


; TenToX(x)- Computes 10**x.
;
;               This routine requires three free registers.
;
;       TenToX(x) = 2**(x * lg(10))

TenToX          proc    near
                fldl2t                  ;Put lg(10) onto the stack
                fmul                    ;Compute x*lg(10)
                call    TwoToX          ;Compute 2**(x * lg(10)).
                ret
TenToX          endp

; exp(x)- Computes e**x.
;
;               This routine requires three free registers.
;
;       exp(x) = 2**(x * lg(e))
```

```
exp             proc    near
                fldl2e                  ;Put lg(e) onto the stack.
                fmul                    ;Compute x*lg(e).
                call    TwoToX          ;Compute 2**(x * lg(e))
                ret
exp             endp

; YtoX(y,x)- Computes y**x (y=st(1), x=st(0)).
;
;               This routine requires three free registers.
;
;               Y must be greater than zero.
;
;       YtoX(y,x) = 2 ** (x * lg(y))

YtoX            proc    near
                fxch                    ;Compute lg(y).
                fld1
                fxch
                fyl2x

                fmul                    ;Compute x*lg(y).
                call    TwoToX          ;Compute 2**(x*lg(y)).
                ret
YtoX            endp

; LOG(x)- Computes the base 10 logarithm of x.
;
;               Usual range for x (>0).
;
;       LOG(x) = lg(x)/lg(10).

log             proc    near
                fld1
                fxch
                fyl2x                   ;Compute 1*lg(x).
                fldl2t                  ;Load lg(10).
                fdiv                    ;Compute lg(x)/lg(10).
                ret
log             endp


; LN(x)- Computes the base e logarithm of x.
;
;               X must be greater than zero.
;
;       ln(x) = lg(x)/lg(e).

ln              proc    near
                fld1
                fxch
                fyl2x                   ;Compute 1*lg(x).
                fldl2e                  ;Load lg(e).
                fdiv                    ;Compute lg(x)/lg(10).
                ret
ln              endp

; This main program tests the various functions in this package.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                finit

; Check to see if cot and acot are working properly.
```

```
                fld     cotVar
                call    cot
                fst     cotRes
                call    acot
                fstp    acotRes

                printff
                byte    "x=%8.5gf, cot(x)=%8.5gf, acot(cot(x)) = %8.5gf\n",0
                dword   cotVar, cotRes, acotRes

; Check to see if csc and acsc are working properly.

                fld     cscVar
                call    csc
                fst     cscRes
                call    acsc
                fstp    acscRes

                printff
                byte    "x=%8.5gf, csc(x)=%8.5gf, acsc(csc(x)) = %8.5gf\n",0
                dword   cscVar, cscRes, acscRes

; Check to see if sec and asec are working properly.

                fld     secVar
                call    sec
                fst     secRes
                call    asec
                fstp    asecRes

                printff
                byte    "x=%8.5gf, sec(x)=%8.5gf, asec(sec(x)) = %8.5gf\n",0
                dword   secVar, secRes, asecRes

; Check to see if sin and asin are working properly.

                fld     sinVar
                fsin
                fst     sinRes
                call    asin
                fstp    asinRes

                printff
                byte    "x=%8.5gf, sin(x)=%8.5gf, asin(sin(x)) = %8.5gf\n",0
                dword   sinVar, sinRes, asinRes

; Check to see if cos and acos are working properly.

                fld     cosVar
                fcos
                fst     cosRes
                call    acos
                fstp    acosRes

                printff
                byte    "x=%8.5gf, cos(x)=%8.5gf, acos(cos(x)) = %8.5gf\n",0
                dword   cosVar, cosRes, acosRes

; Check to see if 2**x and lg(x) are working properly.

                fld     Two2xVar
                call    TwoToX
                fst     Two2xRes
                fld1
                fxch
                fyl2x
                fstp    lgxRes

                printff
                byte    "x=%8.5gf, 2**x =%8.5gf, lg(2**x) = %8.5gf\n",0
```

```
                dword   Two2xVar, Two2xRes, lgxRes

; Check to see if 10**x and l0g(x) are working properly.

                fld     Ten2xVar
                call    TenToX
                fst     Ten2xRes
                call    LOG
                fstp    logRes

                printff
                byte    "x=%8.5gf, 10**x =%8.2gf, log(10**x) = %8.5gf\n",0
                dword   Ten2xVar, Ten2xRes, logRes

; Check to see if exp(x) and ln(x) are working properly.

                fld     expVar
                call    exp
                fst     expRes
                call    ln
                fstp    lnRes

                printff
                byte    "x=%8.5gf, e**x =%8.2gf, ln(e**x) = %8.5gf\n",0
                dword   expVar, expRes, lnRes

; Check to see if y**x is working properly.

                fld     Y2Xy
                fld     Y2Xx
                call    YtoX
                fstp    Y2XRes

                printff
                byte    "x=%8.5gf, y =%8.5gf, y**x = %8.4gf\n",0
                dword   Y2Xx, Y2Xy, Y2XRes

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main

Sample program output:
x= 3.00000, cot(x)=-7.01525, acot(cot(x)) = 3.00000
x= 1.50000, csc(x)= 1.00251, acsc(csc(x)) = 1.50000
x= 0.50000, sec(x)= 1.13949, asec(sec(x)) = 0.50000
x= 0.75000, sin(x)= 0.68163, asin(sin(x)) = 0.75000
x= 0.25000, cos(x)= 0.96891, acos(cos(x)) = 0.25000
x=-2.50000, 2**x  = 0.17677, lg(2**x)     = -2.50000
x= 3.75000, 10**x = 5623.41, log(10**x)   = 3.75000
x= 3.25000, e**x  = 25.79,   ln(e**x)     = 3.25000
x= 3.00000, y     = 3.00000, y**x         = 27.0000
```

14.6 Laboratory Exercises

14.6.1 FPU vs StdLib Accuracy

In this laboratory exercise you will will run two programs that perform 20,000,000 floating point additions. These programs do the first 10,000,000 additions using the 80x87 FPU, they do the second 10,000,000 additions using the Standard Library's floating point routines. This exercise demonstrates the relative accuracy of the two floating point mechanisms.

For your lab report: assemble and run the EX14_1.asm program (it's on the companion CD-ROM). This program adds together 10,000,000 64-bit floating point values and prints their sum. Describe the results in your lab report. Time these operations and report the time difference in your lab report. Note that the *exact* sum these operations should produce is 1.00000010000e+0000.

After running Ex14_1.asm, repeat this process for the Ex14_2.asm file. Ex14_2 differs from Ex14_1 insofar as Ex14_2 lets the Standard Library routines operate on 80-bit memory operands (the FPU cannot operate on 80-bit memory operands, so this part remains unchanged). Time the execution of Ex14_2's two components. Compare these times against the running time of Ex14_1 and explain any differences.

```
; EX14_1.asm
;
; This program runs some tests to determine how well the floating point
; arithmetic in the Standard Library compares with the floating point
; arithmetic on the 80x87.  It does this performing various operations
; using both methods and comparing the result.
;
; Of course, you must have an 80x87 FPU (or 80486 or later processor)
; in order to run this code.

                .386
                option      segment:use16

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment  para public 'data'

; Since this is an accuracy test, this code uses REAL8 values for
; all operations

slValue1        real8    1.0
slSmallVal      real8    1.0e-14

Value1          real8    1.0
SmallVal        real8    1.0e-14

Buffer          byte     20 dup (0)

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                finit                ;Initialize the FPU

; Do 10,000,000 floating point additions:
```

```
                printff
                byte    "Adding 10,000,000 FP values together with the “
                byte    “FPU",cr,lf,0

                mov     ecx, 10000000
FPLoop:         fld     Value1
                fld     SmallVal
                fadd
                fstp    Value1
                dec     ecx
                jnz     FPLoop

                printff
                byte    "Result = %20GE\n",cr,lf,0
                dword   Value1

; Do 10,000,000 floating point additions with the Standard Library fpadd
; routine:

                printff
                byte    cr,lf
                byte    "Adding 10,000,000 FP values together with the “
                byte    “StdLib", cr,lf
                byte    "Note: this may take a few minutes to run, don't “
                byte    “get too impatient"
                byte    cr,lf,0

                mov     ecx, 10000000
SLLoop:         lesi    slValue1
                ldfpa
                lesi    slSmallVal
                ldfpo
                fpadd
                lesi    slValue1
                sdfpa
                dec     ecx
                jnz     SLLoop

                printff
                byte    "Result = %20GE\n",cr,lf,0
                dword   slValue1

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main




; EX14_2.asm
;
; This program runs some tests to determine how well the floating point
; arithmetic in the Standard Library compares with the floating point
; arithmetic on the 80x87.  It lets the standard library routines use
; the full 80-bit format since they allow it and the FPU does not.
;
; Of course, you must have an 80x87 FPU (or 80486 or later processor)
; in order to run this code.
```

```
                .386
                option      segment:use16

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment   para public 'data'

slValue1        real10    1.0
slSmallVal      real10    1.0e-14

Value1          real8     1.0
SmallVal        real8     1.0e-14

Buffer          byte      20 dup (0)

dseg            ends

cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg

Main            proc
                mov       ax, dseg
                mov       ds, ax
                mov       es, ax
                meminit
                finit                 ;Initialize the FPU

; Do 10,000,000 floating point additions:

                printff
                byte      "Adding 10,000,000 FP values together with the “
                byte      “FPU",cr,lf,0

                mov       ecx, 10000000
FPLoop:         fld       Value1
                fld       SmallVal
                fadd
                fstp      Value1
                dec       ecx
                jnz       FPLoop

                printff
                byte      "Result = %20GE\n",cr,lf,0
                dword     Value1

; Do 10,000,000 floating point additions with the Standard Library fpadd
; routine:

                printff
                byte      cr,lf
                byte      "Adding 10,000,000 FP values together with the “
                byte      “StdLib", cr,lf
                byte      "Note: this may take a few minutes to run, don't “
                byte      “get too impatient"
                byte      cr,lf,0

                mov       ecx, 10000000
SLLoop:         lesi      slValue1
                lefpa
                lesi      slSmallVal
                lefpo
                fpadd
                lesi      slValue1
                sefpa
                dec       ecx
                jnz       SLLoop

                printff
```

```
                        byte    "Result = %20LE\n",cr,lf,0
                        dword   slValue1

Quit:                   ExitPgm             ;DOS macro to quit program.
Main                    endp

cseg                    ends

sseg                    segment para stack 'stack'
stk                     db      1024 dup ("stack   ")
sseg                    ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               db      16 dup (?)
zzzzzzseg               ends
                        end     Main
```

14.7 Programming Projects

14.8 Summary

For many applications integer arithmetic has two insurmountable drawbacks – it is not easy to represent fractional values with integers and integers have a limited dynamic range. Floating point arithmetic provides an approximation to real arithmetic that overcomes these two limitations.

Floating point arithmetic, however, is not without its own problems. Floating point arithmetic suffers from *limited precision*. As a result, inaccuracies can creep into a calculation. Therefore, floating point arithmetic does not completely follow normal algebraic rules. There are five very important rules to keep in mind when using floating point arithmetic: (1`) The order of evaluation can affect the accuracy of the result; (2) Whenever adding and subtracting numbers, the accuracy of the result may be less than the precision provided by the floating point format; (3) When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first; (4) When multiplying and dividing values, try to multiply large and small numbers together first and try to divide numbers with the same relative magnitue first; (5) When comparing two floating point numbers, always keep in mind that errors can creep into the computations, therefore you should check to see if one value is within a certain range of the other. For more information, see

- “The Mathematics of Floating Point Arithmetic” on page 771

Early on Intel recognized the need for a hardware floating point unit. They hired three mathematicians to design highly accurate floating point formats and algorithms for their 80x87 family of FPUs. These formats, with slight modifications, become the IEEE 754 and IEEE 854 floating point standards. The IEEE standard actually provides for three different formats: a 32 bit standard precision format, a 64 bit double precision format, and an extended precision format. Intel implemented the extended precision format using 80 bits[9]. The 32 bit format uses a 24 bit mantissa (the H.O. bit is an implied one and is not stored in the 32 bits), an eight bit bias 127 exponent, and a one bit sign. The 64 bit format provides a 53 bit mantissa (again, the H.O. bit is always one and is not stored in the 64 b it value), an 11 bit excess 1023 exponent, and a one bit sign. The 80 bit extended precision format uses a 64 bit exponent, a 15 bit excess 16363 exponent, and a single bit sign. For more information, see

- “IEEE Floating Point Formats” on page 774

9. The IEEE standard only requires that the extended precision format contain more bits than the double precision format.

Although 80x87 FPUs and CPUs with built-in FPUs (80486 and Pentium) are becoming very common, it is still possible that you may need to execute code that uses floating point arithmetic on a machine without an FPU. In such cases you will need to supply software routines to execute the floating point arithmetic. Fortunately, the UCR Standard Library provides a set of floating point routines you can call. The Standard Library includes routines to load and store floating point values, convert between integer and floating point formats, add, subtract, multiply, and divide floating point values, convert between ASCII and floating point, and output floating point values. Even if you have an FPU installed, the Standard Library's conversion and output routines are quite useful. For more information, see

- "The UCR Standard Library Floating Point Routines" on page 777

For fast floating point arithmetic, software doesn't stand a chance against hardware. The 80x87 FPUs provide fast and convient floating point operations by extended the 80x86's instruction set to handle floating point arithmetic. In addition to the new instructions, the 80x87 FPUs also provide eight new data registers, a control register, a status register, and several other internal registers. The FPU data registers, unlike the 80x86's general purpose registers, are organized as a stack. Although it is possible to manipulate the registers as though they were a standard register file, most FPU applications use the stack mechanism when computing floating point results. The FPU control register lets you initialize the 80x87 FPU in one of several different modes. The control register lets you set the rounding control, the precision available during computation, and choose which exceptions can cause an interrupt. The 80x87 status register reports the current state of the FPU. This register provides bits that determine if the FPU is currently busy, determine if a previous instruction has generated an exception, determine the physical register number of the top of the register stack, and provide the FPU condition codes. For more information on the 80x87 register set, see

- "The 80x87 Floating Point Coprocessors" on page 781
- "FPU Registers" on page 781
- "The FPU Data Registers" on page 782
- "The FPU Control Register" on page 782
- "The FPU Status Register" on page 785

In addition to the IEEE single, double, and extended preoision data types, the 80x87 FPUs also support various integer and BCD data types. The FPU will automatically convert to and from these data types when loading and storing such values. For more information on these data type formats, see

- "FPU Data Types" on page 788

The 80x87 FPUs provide a wide range of floating point operations by augmenting the 80x86's instruction set. We can classify the FPU instructions into eight categories: data movement instructions, conversions, arithmetic instructions, comparison instructions, constant instructions, transcendental instructions, miscellaneous instructions, and integer instructions. For more information on these instruction types, see

- "The FPU Instruction Set" on page 789
- "FPU Data Movement Instructions" on page 789
- "Conversions" on page 791
- "Arithmetic Instructions" on page 792
- "Comparison Instructions" on page 797
- "Constant Instructions" on page 798
- "Transcendental Instructions" on page 799
- "Miscellaneous instructions" on page 800
- "Integer Operations" on page 803

Although the 80387 and later FPUs provide a rich set of transcendental functions, there are many trigonometric, inverse trigonometric, exponential, and logarithmic functions missing from the instruction set. However, the missing functions are easy to synthesize using algebraic identities. This chapter provides source code for many of these routines as an example of FPU programming. For more information, see

- “Sample Program: Additional Trigonometric Functions” on page 804

14.9 Questions

1) Why don't the normal rules of algebra apply to floating point arithmetic?

2) Give an example of a sequence of operations whose order of evaluation will produce different results with finite precision arithmetic.

3) Explain why limited precision addition and subtraction operations can cause a loss of precision during a calculation.

4) Why should you, if at all possible, perform multiplications and divisions first in a calculation involving multiplication or division as well as addition or subtraction?

5) Explain the difference between a normalized, unnormalized, and denormalized floating point value.

6) Using the UCR Standard Library, convert the following expression to 80x86 assembly code (assume all variables are 64 bit double precision values). Be sure to perform any necessary algebraic manipulations to ensure the maximum accuracy. You can assume all variables fall in the range ±1e-10…±1e+10.

a) Z := X * X + Y * Y

b) Z := (X-Y)*Z

c) Z := X*Y - X / Y

d) Z := (X+Y) / (X-Y)

e) Z := (X*X) / (Y*Y)

f) Z := X*X + Y + 1.0

7) Convert the above statements to 80x87 FPU code.

8) The following problems provide definitions for the *hyperbolic trigonometric* functions. Encode each of these using the 80x87 FPU instructions and the exp(x) and ln(x) routines provided in this chapter.

a) $\sinh x = \frac{e^x - e^{-x}}{2}$

b) $\cosh x = \frac{e^x + e^{-x}}{2}$

c) $\tanh x = \frac{\sinh x}{\cosh x}$

d) $\operatorname{csch} x = \frac{1}{\sinh x}$

e) $\operatorname{sech} x = \frac{1}{\cosh x}$

f) $\coth x = \frac{\cosh x}{\sinh x}$

g) $\operatorname{asinh} x = ln(x + \sqrt{x^2 + 1})$

h) $\operatorname{acosh} x = ln(x + \sqrt{x^2 - 1})$

i) $\operatorname{atanh} x = \frac{ln(\frac{1+x}{1-x})}{2}$

j) $\operatorname{acsch} x = ln(\frac{x \pm \sqrt{1+x^2}}{x})$

k) $\operatorname{asech} x = ln(\frac{x \pm \sqrt{1-x^2}}{x})$

l) $\operatorname{atanh} x = \frac{ln(\frac{x+1}{x-1})}{2}$

9) Create a log(x,y) function which computes $\log_y x$. The algebraic identity for this is

$$\log_y x = \frac{\log_2 x}{\log_2 y}$$

10) Interval arithmetic involves performing a calculation with every result rounded down and then repeating the computation with each result rounded up. At the end of these two computations, you know that the true result must lie between the two computed results. The rounding control bits in the FPU control register let you select round up and round down modes. Repeat question six applying interval arithmetic and compute the two bounds for each of those problems (a-f).

11) The mantissa precision control bits in the FPU control register simply control where the FPU rounds results. Selecting a lower precision does not improve the performance of the FPU. Therefore, any new software you write should set these two bits to ones to get 64 bits of precision when performing calculations. Can you provide one reason why you might want to set the precision to something other than 64 bits?

12) Suppose you have two 64 bit variables, X and Y, that you want to compare to see if they are equal. As you know, you should not compare them directly to see if they are equal, but rather see if they are less than some small value apart. Suppose ε, the error constant, is 1e-300. Provide the code to load ax with zero if X=Y and load ax with one if X≠Y.

13) Repeat problem 12, except test for:

a) $X \leq Y$ b) $X < Y$

c) $X \geq Y$ d) $X > Y$

e) $X \neq Y$

14) What instruction can you use to see if the value in st(0) is denormalized?

15) Assuming no stack underflow or overflow, what is the C_1 condition code bit usually used for?

16) Many texts, when describing the FPU chip, suggest that you can use the FPU to perform integer arithmetic. An argument generally given is that the FPU can support 64 bit integers whereas the CPU can only support 16 or 32 bit integers. What is wrong with this argument? Why would you *not* want to use the FPU to perform integer arithmetic? Why does the FPU even provide integer instructions?

17) Suppose you have a 64 bit double precision floating point value in memory. Describe how you could take the absolute value of this variable without using the FPU (i.e., by using only 80x86 instructions).

18) Explain how to change the sign of the variable in question 17.

19) Why does the TwoToX function (see “Sample Program: Additional Trigonometric Functions” on page 804) have to compute the result using fscale and fyl2x? Why can’t it use fyl2x along?

20) Explain a possible problem with the following code sequence:

```
stp     mem_64
xor     byte ptr mem_64+7, 80h          ;Tweak sign bit
```

Strings and Character Sets Chapter 15

A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes, words, or (on 80386 and later processors) double words. The 80x86 microprocessor family supports several instructions specifically designed to cope with strings. This chapter explores some of the uses of these string instructions.

The 8088, 8086, 80186, and 80286 can process two types of strings: byte strings and word strings. The 80386 and later processors also handle double word strings. They can move strings, compare strings, search for a specific value within a string, initialize a string to a fixed value, and do other primitive operations on strings. The 80x86's string instructions are also useful for manipulating arrays, tables, and records. You can easily assign or compare such data structures using the string instructions. Using string instructions may speed up your array manipulation code considerably.

15.0 Chapter Overview

This chapter presents a review of the operation of the 80x86 string instructions. Then it discusses how to process character strings using these instructions. Finally, it concludes by discussing the string instruction available in the UCR Standard Library. The sections below that have a "•" prefix are essential. Those sections with a "❑" discuss advanced topics that you may want to put off for a while.

- The 80x86 string instructions.
- Character strings.
- Character string functions.
- String functions in the UCR Standard Library.
- ❑ Using the string instructions on other data types.

15.1 The 80x86 String Instructions

All members of the 80x86 family support five different string instructions: movs, cmps, scas, lods, and stos[1]. They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections.

15.1.1 How the String Instructions Operate

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the movs instruction moves a sequence of bytes from one memory location to another. The cmps instruction compares two blocks of memory. The scas instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the movs instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).

Unlike other instructions which operate on memory, the string instructions are single-byte instructions which don't have any explicit operands. The operands for the string instructions include

1. The 80186 and later processor support two additional string instructions, INS and OUTS which input strings of data from an input port or output strings of data to an output port. We will not consider these instructions in this chapter.

- the si (source index) register,
- the di (destination index) register,
- the cx (count) register,
- the ax register, and
- the direction flag in the FLAGS register.

For example, one variant of the movs (move string) instruction copies a string from the source address specified by ds:si to the destination address specified by es:di, of length cx. Likewise, the cmps instruction compares the string pointed at by ds:si, of length cx, to the string pointed at by es:di.

Not all instructions have source and destination operands (only movs and cmps support them). For example, the scas instruction (scan a string) compares the value in the accumulator to values in memory. Despite their differences, the 80x86's string instructions all have one thing in common – using them requires that you deal with two segments, the data segment and the extra segment.

15.1.2 The REP/REPE/REPZ and REPNZ/REPNE Prefixes

The string instructions, by themselves, do not operate on strings of data. The movs instruction, for example, will move a single byte, word, or double word. When executed by itself, the movs instruction ignores the value in the cx register. The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

```
Field:
Label  repeat    mnemonic    operand              ;comment

For MOVS:
       rep       movs        {operands}

For CMPS:
       repe      cmps        {operands}
       repz      cmps        {operands}
       repne     cmps        {operands}
       repnz     cmps        {operands}

For SCAS:
       repe      scas        {operands}
       repz      scas        {operands}
       repne     scas        {operands}
       repnz     scas        {operands}

For STOS:
       rep       stos        {operands}
```

You don't normally use the repeat prefixes with the lods instruction.

As you can see, the presence of the repeat prefixes introduces a new field in the source line – the repeat prefix field. This field appears only on source lines containing string instructions. In your source file:

- the label field should always begin in column one,
- the repeat field should begin at the first tab stop, and
- the mnemonic field should begin at the second tab stop.

When specifying the repeat prefix before a string instruction, the string instruction repeats cx times[2]. Without the repeat prefix, the instruction operates only on a single byte, word, or double word.

2. Except for the cmps instruction which repeats *at most* the number of times specified in the cx register.

You can use repeat prefixes to process entire strings with a single instruction. You can use the string instructions, without the repeat prefix, as string primitive operations to synthesize more powerful string operations.

The operand field is optional. If present, MASM simply uses it to determine the size of the string to operate on. If the operand field is the name of a byte variable, the string instruction operates on bytes. If the operand is a word address, the instruction operates on words. Likewise for double words. If the operand field is not present, you must append a "B", "W", or "D" to the end of the string instruction to denote the size, e.g., movsb, movsw, or movsd.

15.1.3 The Direction Flag

Besides the si, di, si, and ax registers, one other register controls the 80x86's string instructions – the flags register. Specifically, the *direction flag* in the flags register controls how the CPU processes strings.

If the direction flag is clear, the CPU increments si and di after operating upon each string element. For example, if the direction flag is clear, then executing movs will move the byte, word, or double word at ds:si to es:di and will increment si and di by one, two, or four. When specifying the rep prefix before this instruction, the CPU increments si and di for each element in the string. At completion, the si and di registers will be pointing at the first item beyond the string.

If the direction flag is set, then the 80x86 decrements si and di after processing each string element. After a repeated string operation, the si and di registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the cld (clear direction flag) and std (set direction flag) instructions. When using these instructions inside a procedure, keep in mind that they modify the machine state. Therefore, you may need to save the direction flag during the execution of that procedure. The following example exhibits the kinds of problems you might encounter:

```
StringStuff:
                cld
        <do some operations>
                call     Str2
        <do some string operations requiring D=0>
                  .
                  .
                  .
Str2            proc     near
                std
        <Do some string operations>
                ret
Str2            endp
```

This code will not work properly. The calling code assumes that the direction flag is clear after Str2 returns. However, this isn't true. Therefore, the string operations executed after the call to Str2 will not function properly.

There are a couple of ways to handle this problem. The first, and probably the most obvious, is always to insert the cld or std instructions immediately before executing a string instruction. The other alternative is to save and restore the direction flag using the pushf and popf instructions. Using these two techniques, the code above would look like this:

Always issuing cld or std before a string instruction:

```
StringStuff:
                cld
        <do some operations>
                call     Str2
                cld
        <do some string operations requiring D=0>
```

```
                .
                .
                .
Str2            proc    near
                std
        <Do some string operations>
                ret
Str2            endp
```

Saving and restoring the flags register:

```
StringStuff:
                cld
        <do some operations>
                call    Str2
        <do some string operations requiring D=0>
                .
                .
                .
Str2            proc    near
                pushf
                std
        <Do some string operations>
                popf
                ret
Str2            endp
```

If you use the pushf and popf instructions to save and restore the flags register, keep in mind that you're saving and restoring all the flags. Therefore, such subroutines cannot return any information in the flags. For example, you will not be able to return an error condition in the carry flag if you use pushf and popf.

15.1.4 The MOVS Instruction

The movs instruction takes four basic forms. Movs moves bytes, words, or double words, movsb moves byte strings, movsw moves word strings, and movsd moves double word strings (on 80386 and later processors). These four instructions use the following syntax:

```
{REP} MOVSB
{REP} MOVSW
{REP} MOVSD           ;Available only on 80386 and later processors
{REP} MOVS Dest, Source
```

The movsb (move string, bytes) instruction fetches the byte at address ds:si, stores it at address es:di, and then increments or decrements the si and di registers by one. If the rep prefix is present, the CPU checks cx to see if it contains zero. If not, then it moves the byte from ds:si to es:di and decrements the cx register. This process repeats until cx becomes zero.

The movsw (move string, words) instruction fetches the word at address ds:si, stores it at address es:di, and then increments or decrements si and di by two. If there is a rep prefix, then the CPU repeats this procedure as many times as specified in cx.

The movsd instruction operates in a similar fashion on double words. Incrementing or decrementing si and di by four for each data movement.

MASM automatically figures out the size of the movs instruction by looking at the size of the operands specified. If you've defined the two operands with the byte (or comparable) directive, then MASM will emit a movsb instruction. If you've declared the two labels via word (or comparable), MASM will generate a movws instruction. If you've declared the two labels with dword, MASM emits a movsd instruction. The assembler will also check the segments of the two operands to ensure they match the current assumptions (via the assume directive) about the es and ds registers. You should always use the movsb, movsw, and movsd forms and forget about the movs form.

Although, in theory, the movs form appears to be an elegant way to handle the move string instruction, in practice it creates more trouble than it's worth. Furthermore, this form of the move string instruction implies that movs has explicit operands, when, in fact, the si and di registers implicitly specify the operands. For this reason, we'll always use the movsb, movsw, or movsd instructions. When used with the rep prefix, the movsb instruction will move the number of bytes specified in the cx register. The following code segment copies 384 bytes from String1 to String2:

```
                cld
                lea     si, String1
                lea     di, String2
                mov     cx, 384
        rep     movsb
                  .
                  .
                  .
String1         byte    384 dup (?)
String2         byte    384 dup (?)
```

This code, of course, assumes that String1 and String2 are in the same segment and both the ds and es registers point at this segment. If you substitute movws for movsb, then the code above will move 384 words (768 bytes) rather than 384 bytes:

```
                cld
                lea     si, String1
                lea     di, String2
                mov     cx, 384
        rep     movsw
                  .
                  .
                  .
String1         word    384 dup (?)
String2         word    384 dup (?)
```

Remember, the cx register contains the element count, not the byte count. When using the movsw instruction, the CPU moves the number of words specified in the cx register.

If you've set the direction flag before executing a movsb/movsw/movsd instruction, the CPU decrements the si and di registers after moving each string element. This means that the si and di registers must point at the end of their respective strings before issuing a movsb, movsw, or movsd instruction. For example,

```
                std
                lea     si, String1+383
                lea     di, String2+383
                mov     cx, 384
        rep     movsb
                  .
                  .
                  .
String1         byte    384 dup (?)
String2         byte    384 dup (?)
```

Although there are times when processing a string from tail to head is useful (see the cmps description in the next section), generally you'll process strings in the forward direction since it's more straightforward to do so. There is one class of string operations where being able to process strings in both directions is absolutely mandatory: processing strings when the source and destination blocks overlap. Consider what happens in the following code:

```
                cld
                lea     si, String1
                lea     di, String2
                mov     cx, 384
        rep     movsb
                  .
                  .
                  .
String1         byte    ?
String2         byte    384 dup (?)
```

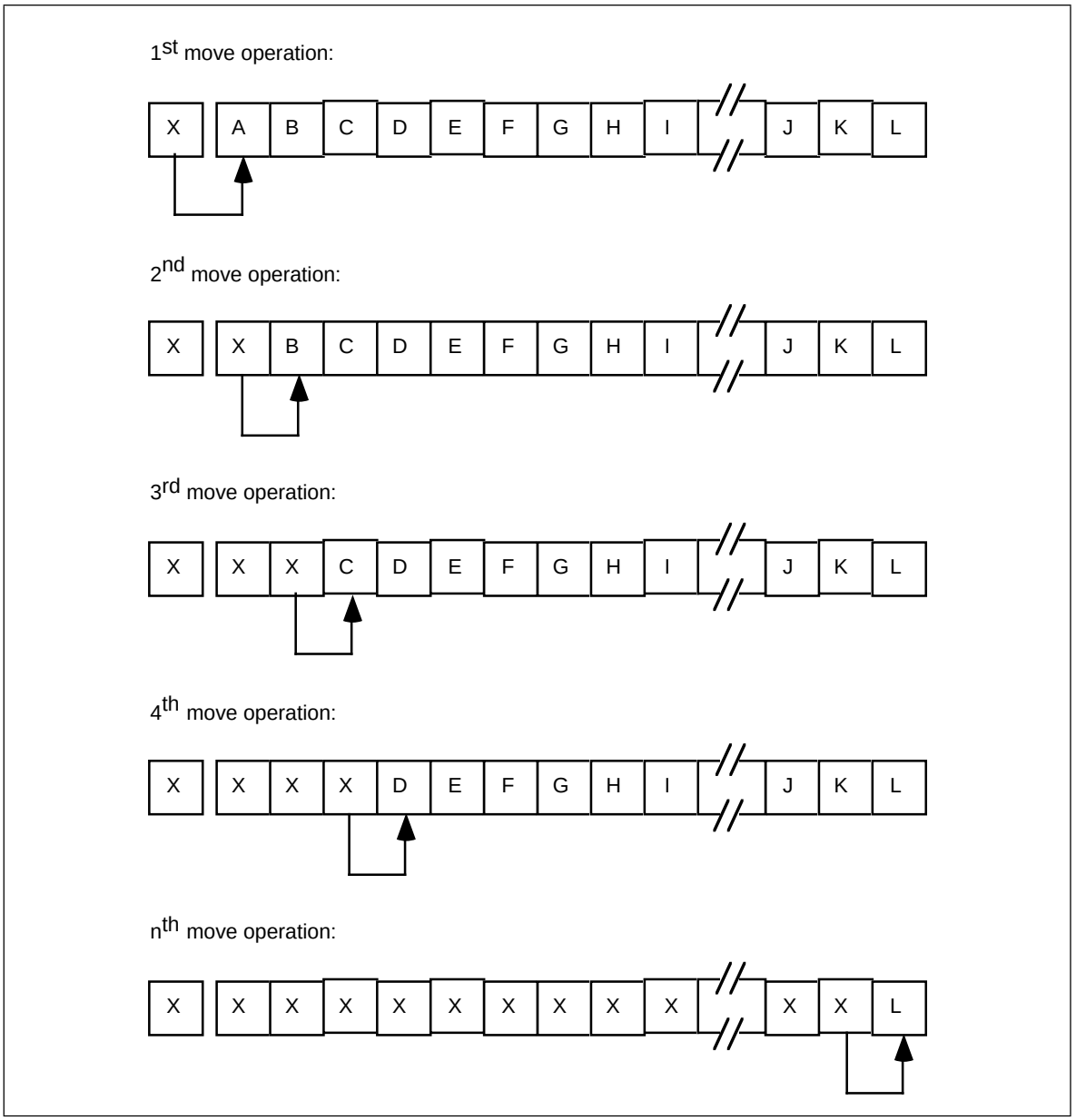


Figure 15.1 Overwriting Data During a Block Move Operation

This sequence of instructions treats String1 and String2 as a pair of 384 byte strings. However, the last 383 bytes in the String1 array overlap the first 383 bytes in the String2 array. Let's trace the operation of this code byte by byte.

When the CPU executes the movsb instruction, it copies the byte at ds:si (String1) to the byte pointed at by es:di (String2). Then it increments si and di, decrements cx by one, and repeats this process. Now the si register points at String1+1 (which is the address of String2) and the di register points at String2+1. The movsb instruction copies the byte pointed at by si to the byte pointed at by di. However, this is the byte originally copied from location String1. So the movsb instruction copies the value originally in location String1 to both locations String2 and String2+1. Again, the CPU increments si and di, decrements cx, and repeats this operation. Now the movsb instruction copies the byte from location String1+2 (String2+1) to location String2+2. But once again, this is the value that originally appeared in location String1. Each repetition of the loop copies the next element in String1 to the next available location in the String2 array. Pictorially, it looks something like that in Figure 15.1.

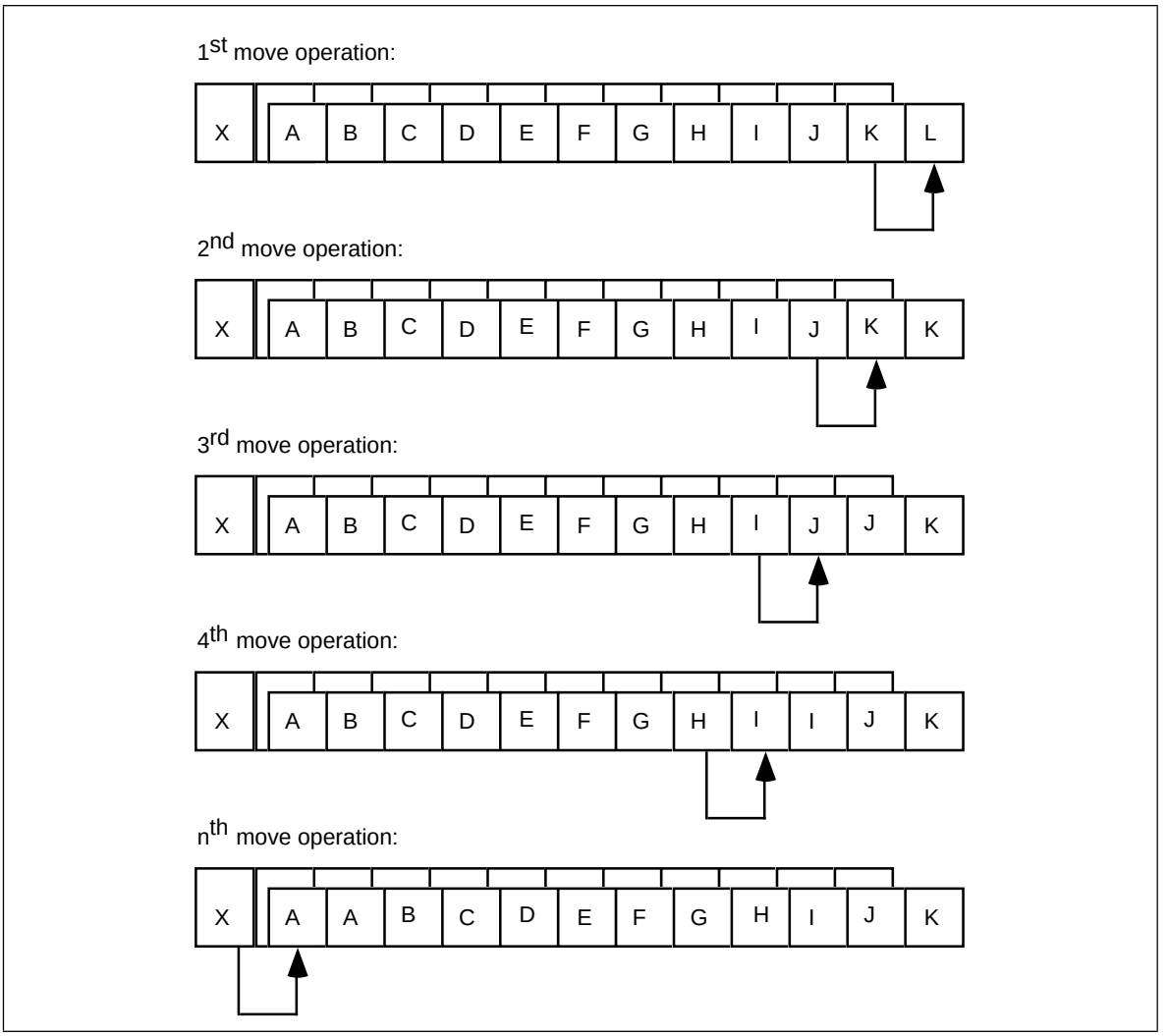


Figure 15.2 Correct Way to Move Data With a Block Move Operation

The end result is that X gets replicated throughout the string. The move instruction copies the source operand into the memory location which will become the source operand for the very next move operation, which causes the replication.

If you really want to move one array into another when they overlap, you should move each element of the source string to the destination string starting at the end of the two strings as shown in Figure 15.2.

Setting the direction flag and pointing si and di at the end of the strings will allow you to (correctly) move one string to another when the two strings overlap and the source string begins at a lower address than the destination string. If the two strings overlap and the source string begins at a higher address than the destination string, then clear the direction flag and point si and di at the beginning of the two strings.

If the two strings do not overlap, then you can use either technique to move the strings around in memory. Generally, operating with the direction flag clear is the easiest, so that makes the most sense in this case.

You shouldn't use the movs instruction to fill an array with a single byte, word, or double word value. Another string instruction, stos, is much better suited for this purpose. However, for arrays whose elements are larger than four bytes, you can use the movs instruction to initialize the entire array to the content of the first element. See the questions for additional information.

15.1.5 The CMPS Instruction

The cmps instruction compares two strings. The CPU compares the string referenced by es:di to the string pointed at by ds:si. Cx contains the length of the two strings (when using the rep prefix). Like the movs instruction, the MASM assembler allows several different forms of this instruction:

```
{REPE}    CMPSB
{REPE}    CMPSW
{REPE}    CMPSD                ;Available only on 80386 and later
{REPE}    CMPS     dest, source
{REPNE}   CMPSB
{REPNE}   CMPSW
{REPNE}   CMPSD                ;Available only on 80386 and later
{REPNE}   CMPS     dest, source
```

Like the movs instruction, the operands present in the operand field of the cmps instruction determine the size of the operands. You specify the actual operand addresses in the si and di registers.

Without a repeat prefix, the cmps instruction subtracts the value at location es:di from the value at ds:si and updates the flags. Other than updating the flags, the CPU doesn't use the difference produced by this subtraction. After comparing the two locations, cmps increments or decrements the si and di registers by one, two, or four (for cmpsb/cmpsw/cmpsd, respectively). Cmps increments the si and di registers if the direction flag is clear and decrements them otherwise.

Of course, you will not tap the real power of the cmps instruction using it to compare single bytes or words in memory. This instruction shines when you use it to compare whole strings. With cmps, you can compare consecutive elements in a string until you find a match or until consecutive elements do not match.

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match. Consider the following strings:

"String1"

"String1"

The only way to determine that these two strings are equal is to compare each character in the first string to the corresponding character in the second. After all, the second string could have been "String2" which definitely is not equal to "String1". Of course, once you encounter a character in the destination string which doesn't equal the corresponding character in the source string, the comparison can stop. You needn't compare any other characters in the two strings.

The repe prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and cx is greater than zero. We could compare the two strings above using the following 80x86 assembly language code:

```
; Assume both strings are in the same segment and ES and DS
; both point at this segment.

                cld
                lea      si, AdrsString1
                lea      di, AdrsString2
                mov      cx, 7
    repe        cmpsb
```

After the execution of the cmpsb instruction, you can test the flags using the standard conditional jump instructions. This lets you check for equality, inequality, less than, greater than, etc.

Character strings are usually compared using *lexicographical ordering*. In lexicographical ordering, the least significant element of a string carries the most weight. This is in direct contrast to standard integer comparisons where the most significant portion of the

number carries the most weight. Furthermore, the length of a string affects the comparison only if the two strings are identical up to the length of the shorter string. For example, “Zebra” is less than “Zebras”, because it is the shorter of the two strings, however, “Zebra” is greater than “AAAAAAAAAAH!” even though it is shorter. Lexicographical comparisons compare corresponding elements until encountering a character which doesn’t match, or until encountering the end of the shorter string. If a pair of corresponding characters do not match, then this algorithm compares the two strings based on that single character. If the two strings match up to the length of the shorter string, we must compare their length. The two strings are equal if and only if their lengths are equal and each corresponding pair of characters in the two strings is identical. Lexicographical ordering is the standard alphabetical ordering you’ve grown up with.

For character strings, use the cmps instruction in the following manner:

- The direction flag must be cleared before comparing the strings.
- Use the cmpsb instruction to compare the strings on a byte by byte basis. Even if the strings contain an even number of characters, you cannot use the cmpsw instruction. It does not compare strings in lexicographical order.
- The cx register must be loaded with the length of the smaller string.
- Use the repe prefix.
- The ds:si and es:di registers must point at the very first character in the two strings you want to compare.

After the execution of the cmps instruction, if the two strings were equal, their lengths must be compared in order to finish the comparison. The following code compares a couple of character strings:

```
                lea     si, source
                lea     di, dest
                mov     cx, lengthSource
                mov     ax, lengthDest
                cmp     cx, ax
                ja      NoSwap
                xchg    ax, cx
NoSwap: repe    cmpsb
                jne     NotEqual
                mov     ax, lengthSource
                cmp     ax, lengthDest
NotEqual:
```

If you’re using bytes to hold the string lengths, you should adjust this code appropriately.

You can also use the cmps instruction to compare multi-word integer values (that is, extended precision integer values). Because of the amount of setup required for a string comparison, this isn’t practical for integer values less than three or four words in length, but for large integer values, it’s an excellent way to compare such values. Unlike character strings, we cannot compare integer strings using a lexicographical ordering. When comparing strings, we compare the characters from the least significant byte to the most significant byte. When comparing integers, we must compare the values from the most significant byte (or word / double word) down to the least significant byte, word or double word. So, to compare two eight-word (128-bit) integer values, use the following code on the 80286:

```
                std
                lea     si, SourceInteger+14
                lea     di, DestInteger+14
                mov     cx, 8
    repe        cmpsw
```

This code compares the integers from their most significant word down to the least significant word. The cmpsw instruction finishes when the two values are unequal or upon decrementing cx to zero (implying that the two values are equal). Once again, the flags provide the result of the comparison.

The repne prefix will instruct the cmps instruction to compare successive string elements as long as they do not match. The 80x86 flags are of little use after the execution of this instruction. Either the cx register is zero (in which case the two strings are totally different), or it contains the number of elements compared in the two strings until a match. While this form of the cmps instruction isn't particularly useful for comparing strings, it is useful for locating the first pair of matching items in a couple of byte or word arrays. In general, though, you'll rarely use the repne prefix with cmps.

One last thing to keep in mind with using the cmps instruction – the value in the cx register determines the number of elements to process, not the number of bytes. Therefore, when using cmpsw, cx specifies the number of words to compare. This, of course, is twice the number of bytes to compare.

15.1.6 The SCAS Instruction

The cmps instruction compares two strings against one another. You cannot use it to search for a particular element within a string. For example, you could not use the cmps instruction to quickly scan for a zero throughout some other string. You can use the scas (scan string) instruction for this task.

Unlike the movs and cmps instructions, the scas instruction only requires a destination string (es:di) rather than both a source and destination string. The source operand is the value in the al (scasb), ax (scasw), or eax (scasd) register.

The scas instruction, by itself, compares the value in the accumulator (al, ax, or eax) against the value pointed at by es:di and then increments (or decrements) di by one, two, or four. The CPU sets the flags according to the result of the comparison. While this might be useful on occasion, scas is a lot more useful when using the repe and repne prefixes.

When the repe prefix (repeat while equal) is present, scas scans the string searching for an element which does not match the value in the accumulator. When using the repne prefix (repeat while not equal), scas scans the string searching for the first string element which is equal to the value in the accumulator.

You're probably wondering "why do these prefixes do exactly the opposite of what they ought to do?" The paragraphs above haven't quite phrased the operation of the scas instruction properly. When using the repe prefix with scas, the 80x86 scans through the string while the value in the accumulator is equal to the string operand. This is equivalent to searching through the string for the first element which does not match the value in the accumulator. The scas instruction with repne scans through the string while the accumulator is not equal to the string operand. Of course, this form searches for the first value in the string which matches the value in the accumulator register. The scas instruction takes the following forms:

```
{REPE}    SCASB
{REPE}    SCASW
{REPE}    SCASD      ;Available only on 80386 and later processors
{REPE}    SCAS     dest
{REPNE}   SCASB
{REPNE}   SCASW
{REPNE}   SCASD      ;Available only on 80386 and later processors
{REPNE}   SCAS     dest
```

Like the cmps and movs instructions, the value in the cx register specifies the number of elements to process, not bytes, when using a repeat prefix.

15.1.7 The STOS Instruction

The stos instruction stores the value in the accumulator at the location specified by es:di. After storing the value, the CPU increments or decrements di depending upon the state of the direction flag. Although the stos instruction has many uses, its primary use is

to initialize arrays and strings to a constant value. For example, if you have a 256-byte array you want to clear out with zeros, use the following code:

```
; Presumably, the ES register already points at the segment
; containing DestString

                cld
                lea     di, DestString
                mov     cx, 128                 ;256 bytes is 128 words.
                xor     ax, ax                  ;AX := 0
        rep     stosw
```

This code writes 128 words rather than 256 bytes because a single stosw operation is faster than two stosb operations. On an 80386 or later this code could have written 64 double words to accomplish the same thing even faster.

The stos instruction takes four forms. They are

```
{REP}   STOSB
{REP}   STOSW
{REP}   STOSD
{REP}   STOS    dest
```

The stosb instruction stores the value in the al register into the specified memory location(s), the stosw instruction stores the ax register into the specified memory location(s) and the stosd instruction stores eax into the specified location(s). The stos instruction is either an stosb, stosw, or stosd instruction depending upon the size of the specified operand.

Keep in mind that the stos instruction is useful only for initializing a byte, word, or dword array to a constant value. If you need to initialize an array to different values, you cannot use the stos instruction. You can use movs in such a situation, see the exercises for additional details.

15.1.8 The LODS Instruction

The lods instruction is unique among the string instructions. You will never use a repeat prefix with this instruction. The lods instruction copies the byte or word pointed at by ds:si into the al, ax, or eax register, after which it increments or decrements the si register by one, two, or four. Repeating this instruction via the repeat prefix would serve no purpose whatsoever since the accumulator register will be overwritten each time the lods instruction repeats. At the end of the repeat operation, the accumulator will contain the last value read from memory.

Instead, use the lods instruction to fetch bytes (lodsb), words (lodsw), or double words (lodsd) from memory for further processing. By using the stos instruction, you can synthesize powerful string operations.

Like the stos instruction, the lods instruction takes four forms:

```
{REP}   LODSB
{REP}   LODSW
{REP}   LODSD                   ;Available only on 80386 and later
{REP}   LODS    dest
```

As mentioned earlier, you'll rarely, if ever, use the rep prefixes with these instructions[3]. The 80x86 increments or decrements si by one, two, or four depending on the direction flag and whether you're using the lodsb, lodsw, or lodsd instruction.

3. They appear here simply because they are allowed. They're not useful, but they are allowed.

15.1.9 Building Complex String Functions from LODS and STOS

The 80x86 supports only five different string instructions: movs, cmps, scas, lods, and stos[4]. These certainly aren't the only string operations you'll ever want to use. However, you can use the lods and stos instructions to easily generate any particular string operation you like. For example, suppose you wanted a string operation that converts all the upper case characters in a string to lower case. You could use the following code:

```
; Presumably, ES and DS have been set up to point at the same
; segment, the one containing the string to convert.

                lea     si, String2Convert
                mov     di, si
                mov     cx, LengthOfString
Convert2Lower:  lodsb                           ;Get next char in str.
                cmp     al, 'A'                 ;Is it upper case?
                jb      NotUpper
                cmp     al, 'Z'
                ja      NotUpper
                or      al, 20h                 ;Convert to lower case.
NotUpper:       stosb                           ;Store into destination.
                loop    Convert2Lower
```

Assuming you're willing to waste 256 bytes for a table, this conversion operation can be sped up somewhat using the xlat instruction:

```
; Presumably, ES and DS have been set up to point at the same
; segment, the one containing the string to be converted.

                cld
                lea     si, String2Convert
                mov     di, si
                mov     cx, LengthOfString
                lea     bx, ConversionTable
Convert2Lower:  lodsb                           ;Get next char in str.
                xlat                            ;Convert as appropriate.
                stosb                           ;Store into destination.
                loop    Convert2Lower
```

The conversion table, of course, would contain the index into the table at each location except at offsets 41h..5Ah. At these locations the conversion table would contain the values 61h..7Ah (i.e., at indexes 'A'..'Z' the table would contain the codes for 'a'..'z').

Since the lods and stos instructions use the accumulator as an intermediary, you can use any accumulator operation to quickly manipulate string elements.

15.1.10 Prefixes and the String Instructions

The string instructions will accept segment prefixes, lock prefixes, and repeat prefixes. In fact, you can specify all three types of instruction prefixes should you so desire. However, due to a bug in the earlier 80x86 chips (pre-80386), you should never use more than a single prefix (repeat, lock, or segment override) on a string instruction unless your code will only run on later processors; a likely event these days. If you absolutely must use two or more prefixes and need to run on an earlier processor, make sure you turn off the interrupts while executing the string instruction.

4. Not counting INS and OUTS which we're ignoring here.

15.2 Character Strings

Since you'll encounter character strings more often than other types of strings, they deserve special attention. The following sections describe character strings and various types of string operations.

15.2.1 Types of Strings

At the most basic level, the 80x86's string instruction only operate upon arrays of characters. However, since most string data types contain an array of characters as a component, the 80x86's string instructions are handy for manipulating that portion of the string.

Probably the biggest difference between a character string and an array of characters is the length attribute. An array of characters contains a fixed number of characters. Never any more, never any less. A character string, however, has a dynamic run-time length, that is, the number of characters contained in the string at some point in the program. Character strings, unlike arrays of characters, have the ability to change their size during execution (within certain limits, of course).

To complicate things even more, there are two generic types of strings: statically allocated strings and dynamically allocated strings. Statically allocated strings are given a fixed, maximum length at program creation time. The length of the string may vary at run-time, but only between zero and this maximum length. Most systems allocate and deallocate dynamically allocated strings in a memory pool when using strings. Such strings may be any length (up to some reasonable maximum value). Accessing such strings is less efficient than accessing statically allocated strings. Furthermore, garbage collection[5] may take additional time. Nevertheless, dynamically allocated strings are much more space efficient than statically allocated strings and, in some instances, accessing dynamically allocated strings is faster as well. Most of the examples in this chapter will use statically allocated strings.

A string with a dynamic length needs some way of keeping track of this length. While there are several possible ways to represent string lengths, the two most popular are length-prefixed strings and zero-terminated strings. A length-prefixed string consists of a single byte or word that contains the length of that string. Immediately following this length value, are the characters that make up the string. Assuming the use of byte prefix lengths, you could define the string "HELLO" as follows:

```
HelloStr        byte    5,"HELLO"
```

Length-prefixed strings are often called Pascal strings since this is the type of string variable supported by most versions of Pascal[6].

Another popular way to specify string lengths is to use zero-terminated strings. A zero-terminated string consists of a string of characters terminated with a zero byte. These types of strings are often called C-strings since they are the type used by the C/C++ programming language. The UCR Standard Library, since it mimics the C standard library, also uses zero-terminated strings.

Pascal strings are much better than C/C++ strings for several reasons. First, computing the length of a Pascal string is trivial. You need only fetch the first byte (or word) of the string and you've got the length of the string. Computing the length of a C/C++ string is considerably less efficient. You must scan the entire string (e.g., using the scasb instruction) for a zero byte. If the C/C++ string is long, this can take a long time. Furthermore, C/C++ strings cannot contain the NULL character. On the other hand, C/C++ strings can be any length, yet require only a single extra byte of overhead. Pascal strings, however,

5. Reclaiming unused storage.
6. At least those versions of Pascal which support strings.

can be no longer than 255 characters when using only a single length byte. For strings longer than 255 bytes, you'll need two bytes to hold the length for a Pascal string. Since most strings are less than 256 characters in length, this isn't much of a disadvantage.

An advantage of zero-terminated strings is that they are easy to use in an assembly language program. This is particularly true of strings that are so long they require multiple source code lines in your assembly language programs. Counting up every character in a string is so tedious that it's not even worth considering. However, you can write a macro which will easily build Pascal strings for you:

```
PString         macro   String
                local   StringLength, StringStart
                byte    StringLength
StringStart     byte    String
StringLength    =       $-StringStart
                endm
                  .
                  .
                  .
                PString   "This string has a length prefix"
```

As long as the string fits entirely on one source line, you can use this macro to generate Pascal style strings.

Common string functions like concatenation, length, substring, index, and others are much easier to write when using length-prefixed strings. So we'll use Pascal strings unless otherwise noted. Furthermore, the UCR Standard library provides a large number of C/C++ string functions, so there is no need to replicate those functions here.

15.2.2 String Assignment

You can easily assign one string to another using the movsb instruction. For example, if you want to assign the length-prefixed string String1 to String2, use the following:

```
; Presumably, ES and DS are set up already

                lea     si, String1
                lea     di, String2
                mov     ch, 0                   ;Extend len to 16 bits.
                mov     cl, String1             ;Get string length.
                inc     cx                      ;Include length byte.
      rep       movsb
```

This code increments cx by one before executing movsb because the length byte contains the length of the string exclusive of the length byte itself.

Generally, string variables can be initialized to constants by using the PString macro described earlier. However, if you need to set a string variable to some constant value, you can write a StrAssign subroutine which assigns the string immediately following the call. The following procedure does exactly that:

```
                include     stdlib.a
                includelib  stdlib.lib

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg, es:dseg, ss:sseg

; String assignment procedure

MainPgm         proc     far
                mov      ax, seg dseg
                mov      ds, ax
                mov      es, ax

                lea      di, ToString
                call     StrAssign
                byte     "This is an example of how the "
```

```
                byte    “StrAssign routine is used”,0
                nop
                ExitPgm
MainPgm         endp

StrAssign       proc    near
                push    bp
                mov     bp, sp
                pushf
                push    ds
                push    si
                push    di
                push    cx
                push    ax
                push    di          ;Save again for use later.
                push    es
                cld

; Get the address of the source string

                mov     ax, cs
                mov     es, ax
                mov     di, 2[bp]   ;Get return address.
                mov     cx, 0ffffh  ;Scan for as long as it takes.
                mov     al, 0       ;Scan for a zero.
       repne    scasb               ;Compute the length of string.
                neg     cx          ;Convert length to a positive #.
                dec     cx          ;Because we started with -1, not 0.
                dec     cx          ;skip zero terminating byte.

; Now copy the strings

                pop     es          ;Get destination segment.
                pop     di          ;Get destination address.
                mov     al, cl      ;Store length byte.
                stosb

; Now copy the source string.

                mov     ax, cs
                mov     ds, ax
                mov     si, 2[bp]
       rep      movsb

; Update the return address and leave:

                inc     si          ;Skip over zero byte.
                mov     2[bp], si

                pop     ax
                pop     cx
                pop     di
                pop     si
                pop     ds
                popf
                pop     bp
                ret
StrAssign       endp

cseg            ends

dseg            segment para public ‘data’
ToString        byte    255 dup (0)
dseg            ends

sseg            segment para stack ‘stack’
                word    256 dup (?)
sseg            ends
                end     MainPgm
```

This code uses the scas instruction to determine the length of the string immediately following the call instruction. Once the code determines the length, it stores this length into the first byte of the destination string and then copies the text following the call to the string variable. After copying the string, this code adjusts the return address so that it points just beyond the zero terminating byte. Then the procedure returns control to the caller.

Of course, this string assignment procedure isn't very efficient, but it's very easy to use. Setting up es:di is all that you need to do to use this procedure. If you need fast string assignment, simply use the movs instruction as follows:

```
; Presumably, DS and ES have already been set up.

                lea     si, SourceString
                lea     di, DestString
                mov     cx, LengthSource
        rep     movsb
                  .
                  .
                  .
SourceString    byte    LengthSource-1
                byte    "This is an example of how the "
                byte    "StrAssign routine is used"
LengthSource    =       $-SourceString

DestString      byte    256 dup (?)
```

Using in-line instructions requires considerably more setup (and typing!), but it is much faster than the StrAssign procedure. If you don't like the typing, you can always write a macro to do the string assignment for you.

15.2.3 String Comparison

Comparing two character strings was already beaten to death in the section on the cmps instruction. Other than providing some concrete examples, there is no reason to consider this subject any further.

Note: all the following examples assume that es and ds are pointing at the proper segments containing the destination and source strings.

Comparing Str1 to Str2:

```
                lea     si, Str1
                lea     di, Str2

; Get the minimum length of the two strings.

                mov     al, Str1
                mov     cl, al
                cmp     al, Str2
                jb      CmpStrs
                mov     cl, Str2

; Compare the two strings.

CmpStrs:        mov     ch, 0
                cld
        repe    cmpsb
                jne     StrsNotEqual

; If CMPS thinks they're equal, compare their lengths
; just to be sure.

                cmp     al, Str2
StrsNotEqual:
```

At label StrsNotEqual, the flags will contain all the pertinent information about the ranking of these two strings. You can use the conditional jump instructions to test the result of this comparison.

15.3 Character String Functions

Most high level languages, like Pascal, BASIC, "C", and PL/I, provide several string functions and procedures (either built into the language or as part of a standard library). Other than the five string operations provided above, the 80x86 doesn't support any string functions. Therefore, if you need a particular string function, you'll have to write it yourself. The following sections describe many of the more popular string functions and how to implement them in assembly language.

15.3.1 Substr

The Substr (substring) function copies a portion of one string to another. In a high level language, this function usually takes the form:

```
DestStr := Substr(SrcStr,Index,Length);
```

where:

- DestStr is the name of the string variable where you want to store the substring,
- SrcStr is the name of the source string (from which the substring is to be taken),
- Index is the starting character position within the string (1..length(SrcStr)), and
- Length is the length of the substring you want to copy into DestStr.

The following examples show how Substr works.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,11,7);
write(DestStr);
```

This prints 'example'. The index value is eleven, so, the Substr function will begin copying data starting at the eleventh character in the string. The eleventh character is the 'e' in 'example'. The length of the string is seven.

This invocation copies the seven characters 'example' to DestStr.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,1,10);
write(DestStr);
```

This prints 'This is an'. Since the index is one, this occurrence of the Substr function starts copying 10 characters starting with the first character in the string.

```
SrcStr := 'This is an example of a string';
DestStr := Substr(SrcStr,20,11);
write(DestStr);
```

This prints 'of a string'. This call to Substr extracts the last eleven characters in the string.

What happens if the index and length values are out of bounds? For example, what happens if Index is zero or is greater than the length of the string? What happens if Index is fine, but the sum of Index and Length is greater than the length of the source string? You can handle these abnormal situations in one of three ways: (1)ignore the possibility of error; (2)abort the program with a run-time error; (3)process some reasonable number of characters in response to the request.

The first solution operates under the assumption that the caller never makes a mistake computing the values for the parameters to the Substr function. It blindly assumes that the values passed to the Substr function are correct and processes the string based on that assumption. This can produce some bizarre effects. Consider the following examples, which use length-prefixed strings:

```
SourceStr :='1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ';
DestStr := Substr(SourceStr,0,5);
Write('DestStr');
```

prints '$1234'. The reason, of course, is that SourceStr is a length-prefixed string. Therefore the length, 36, appears at offset zero within the string. If Substr uses the illegal index of zero then the length of the string will be returned as the first character. In this particular case, the length of the string, 36, just happened to correspond to the ASCII code for the '$' character.

The situation is considerably worse if the value specified for Index is negative or is greater than the length of the string. In such a case, the Substr function would be returning a substring containing characters appearing before or after the source string. This is not a reasonable result.

Despite the problems with ignoring the possibility of error in the Substr function, there is one big advantage to processing substrings in this manner: the resulting Substr code is more efficient if it doesn't have to perform any run-time checking on the data. If you know that the index and length values are always within an acceptable range, then there is no need to do this checking within Substr function. If you can guarantee that an error will not occur, your programs will run (somewhat) faster by eliminating the run-time check.

Since most programs are rarely error-free, you're taking a big gamble if you assume that all calls to the Substr routine are passing reasonable values. Therefore, some sort of run-time check is often necessary to catch errors in your program. An error occurs under the following conditions:

- The index parameter (Index) is less than one.
- Index is greater than the length of the string.
- The Substr length parameter (Length) is greater than the length of the string.
- The sum of Index and Length is greater than the length of the string.

An alternative to ignoring any of these errors is to abort with an error message. This is probably fine during the program development phase, but once your program is in the hands of users it could be a real disaster. Your customers wouldn't be very happy if they'd spent all day entering data into a program and it aborted, causing them to lose the data they've entered. An alternative to aborting when an error occurs is to have the Substr function return an error condition. Then leave it up to the calling code to determine if an error has occurred. This technique works well with the third alternative to handling errors: processing the substring as best you can.

The third alternative, handling the error as best you can, is probably the best alternative. Handle the error conditions in the following manner:

- The index parameter (Index) is less than one. There are two ways to handle this error condition. One way is to automatically set the Index parameter to one and return the substring beginning with the first character of the source string. The other alternative is to return the *empty string*, a string of length zero, as the substring. Variations on this theme are also possible. You might return the substring beginning with the first character if the index is zero and an empty string if the index is negative. Another alternative is to use unsigned numbers. Then you've only got to worry about the case where Index is zero. A negative number, should the calling code accidentally generate one, would look like a large positive number.

- The index is greater than the length of the string. If this is the case, then the Substr function should return an empty string. Intuitively, this is the proper response in this situation.
- The Substr length parameter (Length) is greater than the length of the string. -or-
- The sum of Index and Length is greater than the length of the string. Points three and four are the same problem, the length of the desired substring extends beyond the end of the source string. In this event, Substr should return the substring consisting of those characters starting at Index through the end of the source string.

The following code for the Substr function expects four parameters: the addresses of the source and destination strings, the starting index, and the length of the desired substring. Substr expects the parameters in the following registers:

ds:si-	The address of the source string.
es:di-	The address of the destination string.
ch-	The starting index.
cl-	The length of the substring.

Substr returns the following values:

- The substring, at location es:di.
- Substr clears the carry flag if there were no errors. Substr sets the carry flag if there was an error.
- Substr preserves all the registers.

If an error occurs, then the calling code must examine the values in si, di and cx to determine the exact cause of the error (if this is necessary). In the event of an error, the Substr function returns the following substrings:

- If the Index parameter (ch) is zero, Substr uses one instead.
- The Index and Length parameters are both unsigned byte values, therefore they are never negative.
- If the Index parameter is greater than the length of the source string, Substr returns an empty string.
- If the sum of the Index and Length parameters is greater than the length of the source string, Substr returns only those characters from Index through the end of the source string. The following code realizes the substring function.

```
; Substring function.
;
; HLL form:
;
;procedure substring(var Src:string;
;                        Index, Length:integer;
;                        var Dest:string);
;
; Src- Address of a source string.
; Index- Index into the source string.
; Length- Length of the substring to extract.
; Dest- Address of a destination string.
;
; Copies the source string from address [Src+index] of length
; Length to the destination string.
;
; If an error occurs, the carry flag is returned set, otherwise
; clear.
;
; Parameters are passed as follows:
;
; DS:SI- Source string address.
; ES:DI- Destination string address.
```

```
; CH- Index into source string.
; CL- Length of source string.
;
; Note: the strings pointed at by the SI and DI registers are
; length-prefixed strings. That is, the first byte of each
; string contains the length of that string.

Substring       proc    near
                push    ax
                push    cx
                push    di
                push    si
                clc                 ;Assume no error.
                pushf               ;Save direction flag status.

; Check the validity of the parameters.

                cmp     ch, [si]    ;Is index beyond the length of
                ja      ReturnEmpty ; the source string?
                mov     al, ch      ;See if the sum of index and
                dec     al          ; length is beyond the end of the
                add     al, cl      ; string.
                jc      TooLong     ;Error if > 255.
                cmp     al, [si]    ;Beyond the length of the source?
                jbe     OkaySoFar

; If the substring isn't completely contained within the source
; string, truncate it:

TooLong:        popf
                stc                 ;Return an error flag.
                pushf
                mov     al, [si]    ;Get maximum length.
                sub     al, ch      ;Subtract index value.
                inc     al          ;Adjust as appropriate.
                mov     cl, al      ;Save as new length.

OkaySoFar:      mov     es:[di], cl ;Save destination string length.
                inc     di
                mov     al, ch      ;Get index into source.
                mov     ch, 0       ;Zero extend length value into CX.
                mov     ah, 0       ;Zero extend index into AX.
                add     si, ax      ;Compute address of substring.
                cld
        rep     movsb               ;Copy the substring.

                popf
SubStrDone:     pop     si
                pop     di
                pop     cx
                pop     ax
                ret

; Return an empty string here:

ReturnEmpty:    mov     byte ptr es:[di], 0
                popf
                stc
                jmp     SubStrDone
SubString       endp
```

15.3.2 Index

The Index string function searches for the first occurrence of one string within another and returns the offset to that occurrence. Consider the following HLL form:

```
SourceStr := 'Hello world';
TestStr := 'world';
I := INDEX(SourceStr, TestStr);
```

The Index function scans through the source string looking for the first occurrence of the test string. If found, it returns the index into the source string where the test string begins. In the example above, the Index function would return seven since the substring 'world' starts at the seventh character position in the source string.

The only possible error occurs if Index cannot find the test string in the source string. In such a situation, most implementations return zero. Our version will do likewise. The Index function which follows operates in the following fashion:

1) It compares the length of the test string to the length of the source string. If the test string is longer, Index immediately returns zero since there is no way the test string will be found in the source string in this situation.

2) The index function operates as follows:

```
i := 1;
while (i < (length(source)-length(test)) and
           test <> substr(source, i, length(test)) do
                     i := i+1;
```

When this loop terminates, if (i < length(source)-length(test)) then it contains the index into source where test begins. Otherwise test is not a substring of source. Using the previous example, this loop compares test to source in the following manner:

```
i=1
test:           world                          No match
source:         Hello world

i=2
test:            world                         No match
source:         Hello world

i=3
test:             world                        No match
source:         Hello world

i=4
test:                world                     No match
source:         Hello world

i=5
test:                world                     No match
source:         Hello world

i=6
test:                 world                    No match
source:         Hello world

i=7
test:                   world                  Match
source:         Hello world
```

There are (algorithmically) better ways to do this comparison[7], however, the algorithm above lends itself to the use of 80x86 string instructions and is very easy to understand. Index's code follows:

```
; INDEX- computes the offset of one string within another.
;
; On entry:
;
```

7. The interested reader should look up the Knuth-Morris-Pratt algorithm in "Data Structure Techniques" by Thomas A. Standish. The Boyer-Moore algorithm is another fast string search routine, although somewhat more complex.

```
; ES:DI-                 Points at the test string that INDEX will search for
;                        in the source string.
; DS:SI-                 Points at the source string which (presumably)
;                         contains the string INDEX is searching for.
;
; On exit:
;
; AX-                    Contains the offset into the source string where the
;                        test string was found.

INDEX           proc    near
                push    si
                push    di
                push    bx
                push    cx
                pushf                   ;Save direction flag value.
                cld

                mov     al, es:[di]     ;Get the length of the test string.
                cmp     al, [si]        ;See if it is longer than the length
                ja      NotThere        ; of the source string.

; Compute the index of the last character we need to compare the
; test string against in the source string.

                mov     al, es:[di]     ;Length of test string.
                mov     cl, al          ;Save for later.
                mov     ch, 0
                sub     al, [si]        ;Length of source string.
                mov     bl, al          ;# of times to repeat loop.
                inc     di              ;Skip over length byte.
                xor     ax, ax          ;Init index to zero.
CmpLoop:        inc     ax              ;Bump index by one.
                inc     si              ;Move on to the next char in source.
                push    si              ;Save string pointers and the
                push    di              ; length of the test string.
                push    cx
       rep      cmpsb                   ;Compare the strings.
                pop     cx              ;Restore string pointers
                pop     di              ; and length.
                pop     si
                je      Foundindex      ;If we found the substring.
                dec     bl
                jnz     CmpLoop         ;Try next entry in source string.

; If we fall down here, the test string doesn't appear inside the
; source string.

NotThere:       xor     ax, ax          ;Return INDEX = 0

; If the substring was found in the loop above, remove the
; garbage left on the stack

FoundIndex:     popf
                pop     cx
                pop     bx
                pop     di
                pop     si
                ret
INDEX           endp
```

15.3.3 Repeat

The Repeat string function expects three parameters– the address of a string, a length, and a character. It constructs a string of the specified length containing "length" copies of

the specified character. For example, Repeat(STR,5,'*') stores the string '*****' into the STR string variable. This is a very easy string function to write, thanks to the stosb instruction:

```
; REPEAT-                 Constructs a string of length CX where each element
;                         is initialized to the character passed in AL.
;
; On entry:
;
; ES:DI-                  Points at the string to be constructed.
; CX-                     Contains the length of the string.
; AL-                     Contains the character with which each element of
;                         the string is to be initialized.

REPEAT          proc    near
                push    di
                push    ax
                push    cx
                pushf                   ;Save direction flag value.
                cld
                mov     es:[di], cl     ;Save string length.
                mov     ch, 0           ;Just in case.
                inc     di              ;Start string at next location.
        rep     stosb
                popf
                pop     cx
                pop     ax
                pop     di
                ret
REPEAT          endp
```

15.3.4 Insert

The Insert string function inserts one string into another. It expects three parameters, a source string, a destination string, and an index. Insert inserts the source string into the destination string starting at the offset specified by the index parameter. HLLs usually call the Insert procedure as follows:

```
source := ' there';
dest := 'Hello world';
INSERT(source,dest,6);
```

The call to Insert above would change source to contain the string 'Hello there world'. It does this by inserting the string ' there' before the sixth character in 'Hello world'.

The insert procedure using the following algorithm:

Insert(Src,dest,index);

1) Move the characters from location dest+index through the end of the destination string length (Src) bytes up in memory.

2) Copy the characters from the Src string to location dest+index.

3) Adjust the length of the destination string so that it is the sum of the destination and source lengths. The following code implements this algorithm:

```
; INSERT- Inserts one string into another.
;
; On entry:
;
; DS:SI Points at the source string to be inserted
;
; ES:DI Points at the destination string into which the source
; string will be inserted.
;
; DX Contains the offset into the destination string where the
```

```
; source string is to be inserted.
;
;
; All registers are preserved.
;
; Error condition-
;
; If the length of the newly created string is greater than 255,
; the insert operation will not be performed and the carry flag
; will be returned set.
;
; If the index is greater than the length of the destination
; string,
; then the source string will be appended to the end of the destin- ; ation
string.

INSERT          proc    near
                push    si
                push    di
                push    dx
                push    cx
                push    bx
                push    ax
                clc                     ;Assume no error.
                pushf
                mov     dh, 0           ;Just to be safe.

; First, see if the new string will be too long.

                mov     ch, 0
                mov     ah, ch
                mov     bh, ch
                mov     al, es:[di]     ;AX = length of dest string.
                mov     cl, [si]        ;CX = length of source string.
                mov     bl, al          ;BX = length of new string.
                add     bl, cl
                jc      TooLong         ;Abort if too long.
                mov     es:[di], bl     ;Update length.

; See if the index value is too large:

                cmp     dl, al
                jbe     IndexIsOK
                mov     dl, al
IndexIsOK:

; Now, make room for the string that's about to be inserted.

                push    si              ;Save for later.
                push    cx

                mov     si, di          ;Point SI at the end of current
                add     si, ax          ; destination string.
                add     di, bx          ;Point DI at the end of new str.
                std
        rep     movsb                   ;Open up space for new string.

; Now, copy the source string into the space opened up.

                pop     cx
                pop     si
                add     si, cx          ;Point at end of source string.
        rep     movsb
                jmp     INSERTDone

TooLong:        popf
                stc
                pushf

INSERTDone:     popf
```

```
                pop     ax
                pop     bx
                pop     cx
                pop     dx
                pop     di
                pop     si
                ret
INSERT          endp
```

15.3.5 Delete

The Delete string removes characters from a string. It expects three parameters – the address of a string, an index into that string, and the number of characters to remove from that string. A HLL call to Delete usually takes the form:

```
Delete(Str,index,length);
```

For example,

```
Str := 'Hello there world';
Delete(str,7,6);
```

This call to Delete will leave str containing 'Hello world'. The algorithm for the delete operation is the following:

1) Subtract the length parameter value from the length of the destination string and update the length of the destination string with this new value.

2) Copy any characters following the deleted substring over the top of the deleted substring.

There are a couple of errors that may occur when using the delete procedure. The index value could be zero or larger than the size of the specified string. In this case, the Delete procedure shouldn't do anything to the string. If the sum of the index and length parameters is greater than the length of the string, then the Delete procedure should delete all the characters to the end of the string. The following code implements the Delete procedure:

```
; DELETE - removes some substring from a string.
;
; On entry:
;
; DS:SI                 Points at the source string.
; DX                    Index into the string of the start of the substring
;                       to delete.
; CX                    Length of the substring to be deleted.
;
; Error conditions-
;
; If DX is greater than the length of the string, then the
; operation is aborted.
;
; If DX+CX is greater than the length of the string, DELETE only
; deletes those characters from DX through the end of the string.

DELETE          proc    near
                push    es
                push    si
                push    di
                push    ax
                push    cx
                push    dx
                pushf                   ;Save direction flag.
                mov     ax, ds          ;Source and destination strings
                mov     es, ax          ; are the same.
                mov     ah, 0
```

```
                mov     dh, ah      ;Just to be safe.
                mov     ch, ah

; See if any error conditions exist.

                mov     al, [si]    ;Get the string length
                cmp     dl, al      ;Is the index too big?
                ja      TooBig
                mov     al, dl      ;Now see if INDEX+LENGTH
                add     al, cl      ;is too large
                jc      Truncate
                cmp     al, [si]
                jbe     LengthIsOK

; If the substring is too big, truncate it to fit.

Truncate:       mov     cl, [si]    ;Compute maximum length
                sub     cl, dl
                inc     cl

; Compute the length of the new string.

LengthIsOK:     mov     al, [si]
                sub     al, cl
                mov     [si], al

; Okay, now delete the specified substring.

                add     si, dx      ;Compute address of the substring
                mov     di, si      ; to be deleted, and the address of
                add     di, cx      ; the first character following it.
                cld
        rep     movsb               ;Delete the string.

TooBig:         popf
                pop     dx
                pop     cx
                pop     ax
                pop     di
                pop     si
                pop     es
                ret
DELETE          endp
```

15.3.6 Concatenation

The concatenation operation takes two strings and appends one to the end of the other. For example, Concat('Hello ','world') produces the string 'Hello world'. Some high level languages treat concatenation as a function call, others as a procedure call. Since in assembly language everything is a procedure call anyway, we'll adopt the procedural syntax. Our Concat procedure will take the following form:

```
Concat(source1,source2,dest);
```

This procedure will copy source1 to dest, then it will concatenate source2 to the end of dest. Concat follows:

```
; Concat-              Copies the string pointed at by SI to the string
;                      rointed at byDI and then concatenates the string;
;                      pointed at by BX to the destination string.
;
; On entry-
;
; DS:SI-               Points at the first source string
; DS:BX-               Points at the second source string
; ES:DI-               Points at the destination string.
```

```
;
; Error condition-
;
; The sum of the lengths of the two strings is greater than 255.
; In this event, the second string will be truncated so that the
; entire string is less than 256 characters in length.

CONCAT          proc    near
                push    si
                push    di
                push    cx
                push    ax
                pushf

; Copy the first string to the destination string:

                mov     al, [si]
                mov     cl, al
                mov     ch, 0
                mov     ah, ch
                add     al, [bx]        ;Compute the sum of the string's
                adc     ah, 0           ; lengths.
                cmp     ax, 256
                jb      SetNewLength
                mov     ah, [si]        ;Save original string length.
                mov     al, 255         ;Fix string length at 255.
SetNewLength:   mov     es:[di], al     ;Save new string length.
                inc     di              ;Skip over length bytes.
                inc     si
       rep      movsb                   ;Copy source1 to dest string.

; If the sum of the two strings is too long, the second string
; must be truncated.
                mov     cl, [bx]        ;Get length of second string.
                cmp     ax, 256
                jb      LengthsAreOK
                mov     cl, ah          ;Compute truncated length.
                neg     cl              ;CL := 256-Length(Str1).

LengthsAreOK:   lea     si, 1[bx]       ;Point at second string and
;                                       ; skip the string length.
                cld
                rep     movsb                   ;Perform the concatenation.

                popf
                pop     ax
                pop     cx
                pop     di
                pop     si
                ret
CONCAT          endp
```

15.4 String Functions in the UCR Standard Library

The UCR Standard Library for 80x86 Assembly Language Programmers provides a very rich set of string functions you may use. These routines, for the most part, are quite similar to the string functions provided in the C Standard Library. As such, these functions support zero terminated strings rather than the length prefixed strings supported by the functions in the previous sections.

Because there are so many different UCR StdLib string routines and the sources for all these routines are in the public domain (and are present on the companion CD-ROM for this text), the following sections will not discuss the implementation of each routine. Instead, the following sections will concentrate on how to *use* these library routines.

The UCR library often provides several variants of the same routine. Generally a suffix of “l”, “m”, or “ml” appears at the end of the name of these variant routines. The “l” suffix stands for “literal constant”. Routines with the “l” (or “ml”) suffix require two string operands. The first is generally pointed at by es:di and the second immediate follows the call in the code stream.

Most StdLib string routines operate on the specified string (or one of the strings if the function has two operands). The “m” (or “ml”) suffix instructs the string function to allocate storage on the heap (using malloc, hence the “m” suffix) for the new string and store the modified result there rather than changing the source string(s). These routines always return a pointer to the newly created string in the es:di registers. In the event of a memory allocation error (insufficient memory), these routines with the “m” or “ml” suffix return the carry flag set. They return the carry clear if the operation was successful.

15.4.1 StrBDel, StrBDelm

These two routines delete leading spaces from a string. StrBDel removes any leading spaces from the string pointed at by es:di. It actually modifies the source string. StrBDelm makes a copy of the string on the heap with any leading spaces removed. If there are no leading spaces, then the StrBDel routines return the original string without modification. Note that these routines only affect *leading* spaces (those appearing at the beginning of the string). They do not remove trailing spaces and spaces in the middle of the string. See Strtrim if you want to remove trailing spaces. Examples:

```
MyString        byte    “    Hello there, this is my string”,0
MyStrPtr        dword   MyString
                  .
                  .
                  .
                les     di, MyStrPtr
                strbdelm        ;Creates a new string w/o leading spaces,
                jc      error ; pointer to string is in ES:DI on return.
                puts            ;Print the string pointed at by ES:DI.
                free            ;Deallocate storage allocated by strbdelm.
                  .
                  .
                  .
; Note that “MyString” still contains the leading spaces.
; The following printf call will print the string along with
; those leading spaces. “strbdelm” above did not change MyString.

                printf
                byte    “MyString = ‘%s’\n”,0
                dword   MyString
                  .
                  .
                  .
                les     di, MyStrPtr
                strbdel

; Now, we really have removed the leading spaces from “MyString”

                printf
                byte    “MyString = ‘%s’\n”,0
                dword   MyString
                  .
                  .
                  .
```

Output from this code fragment:

```
Hello there, this is my string
MyString = ‘   Hello there, this is my string’
MyString = ‘Hello there, this is my string’
```

15.4.2 Strcat, Strcatl, Strcatm, Strcatml

The strcat(xx) routines perform string concatenation. On entry, es:di points at the first string, and for strcat/strcatm dx:si points at the second string. For strcatl and strcatlm the second string follows the call in the code stream. These routines create a new string by appending the second string to the end of the first. In the case of strcat and strcatl, the second string is directly appended to the end of the first string (es:di) in memory. You must make sure there is sufficient memory at the end of the first string to hold the appended characters. Strcatm and strcatml create a new string on the heap (using malloc) holding the concatenated result. Examples:

```
String1         byte    "Hello ",0
                byte    16 dup (0)              ;Room for concatenation.

String2         byte    "world",0

; The following macro loads ES:DI with the address of the
; specified operand.

lesi            macro   operand
                mov     di, seg operand
                mov     es, di
                mov     di, offset operand
                endm

; The following macro loads DX:SI with the address of the
; specified operand.

ldxi            macro   operand
                mov     dx, seg operand
                mov     si, offset operand
                endm
                  .
                  .
                  .
                lesi    String1
                ldxi    String2
                strcatm                 ;Create "Hello world"
                jc      error           ;If insufficient memory.
                print
                byte    "strcatm: ",0
                puts                    ;Print "Hello world"
                putcr
                free                    ;Deallocate string storage.
                  .
                  .
                  .
                lesi    String1         ;Create the string
                strcatml                ; "Hello there"
                jc      error           ;If insufficient memory.
                byte    "there",0
                print
                byte    "strcatml: ",0
                puts                    ;Print "Hello there"
                putcr
                free
                  .
                  .
                  .
                lesi    String1
                ldxi    String2
                strcat                  ;Create "Hello world"
                printf
                byte    "strcat: %s\n",0
                  .
                  .
                  .
; Note: since strcat above has actually modified String1,
; the following call to strcatl appends "there" to the end
; of the string "Hello world".

                lesi    String1
```

```
                strcatl
                byte    "there",0
                printf
                byte    "strcatl: %s\n",0
                  .
                  .
                  .
```

The code above produces the following output:

```
strcatm: Hello world
strcatml: Hello there
strcat: Hello world
strcatl: Hello world there
```

15.4.3 Strchr

Strchr searches for the first occurrence of a single character within a string. In operation it is quite similar to the scasb instruction. However, you do not have to specify an explicit length when using this function as you would for scasb.

On entry, es:di points at the string you want to search through, al contains the value to search for. On return, the carry flag denotes success (C=1 means the character was *not* present in the string, C=0 means the character was present). If the character was found in the string, cx contains the index into the string where strchr located the character. Note that the first character of the string is at index zero. So strchr will return zero if al matches the first character of the string. If the carry flag is set, then the value in cx has no meaning. Example:

```
; Note that the following string has a period at location
; "HasPeriod+24".

HasPeriod       byte    "This string has a period.",0
                  .
                  .
                  .
                lesi    HasPeriod   ;See strcat for lesi definition.
                mov     al, "."     ;Search for a period.
                strchr
                jnc     GotPeriod
                print
                byte    "No period in string",cr,lf,0
                jmp     Done

; If we found the period, output the offset into the string:

GotPeriod:      print
                byte    "Found period at offset ",0
                mov     ax, cx
                puti
                putcr
Done:
```

This code fragment produces the output:

```
Found period at offset 24
```

15.4.4 Strcmp, Strcmpl, Stricmp, Stricmpl

These routines compare strings using a lexicographical ordering. On entry to strcmp or stricmp, es:di points at the first string and dx:si points at the second string. Strcmp compares the first string to the second and returns the result of the comparison in the flags register. Strcmpl operates in a similar fashion, except the second string follows the call in the code stream. The stricmp and stricmpl routines differ from their counterparts in that they ignore case during the comparison. Whereas strcmp would return 'not equal' when comparing "Strcmp" with "strcmp", the stricmp (and stricmpl) routines would return "equal" since the

only differences are upper vs. lower case. The “i” in stricmp and stricmpl stands for “ignore case.” Examples:

```
String1         byte    “Hello world”, 0
String2         byte    “hello world”, 0
String3         byte    “Hello there”, 0
                  .
                  .
                  .
                lesi    String1     ;See strcat for lesi definition.
                ldxi    String2     ;See strcat for ldxi definition.
                strcmp
                jae     IsGtrEql
                printf
                byte    “%s is less than %s\n”,0
                dword   String1, String2
                jmp     Tryl

IsGtrEql:       printf
                byte    “%s is greater or equal to %s\n”,0
                dword   String1, String2

Tryl:           lesi    String2
                strcmpl
                byte    “hi world!”,0
                jne     NotEql
                printf
                byte    “Hmmm..., %s is equal to ‘hi world!’\n”,0
                dword   String2
                jmp     Tryi

NotEql:         printf
                byte    “%s is not equal to ‘hi world!’\n”,0
                dword   String2

Tryi:           lesi    String1
                ldxi    String2
                stricmp
                jne     BadCmp
                printf
                byte    “Ignoring case, %s equals %s\n”,0
                dword   String1, String2
                jmp     Tryil

BadCmp:         printf
                byte    “Wow, stricmp doesn’t work! %s <> %s\n”,0
                dword   String1, String2

Tryil:          lesi    String2
                stricmpl
                byte    “hELLO THERE”,0
                jne     BadCmp2
                print
                byte    “Stricmpl worked”,cr,lf,0
                jmp     Done

BadCmp2:        print
                byte    “Stricmp did not work”,cr,lf,0

Done:
```

15.4.5 Strcpy, Strcpyl, Strdup, Strdupl

The strcpy and strdup routines copy one string to another. There is no strcpym or strcpyml routines. Strdup and strdupl correspond to those operations. The UCR Standard Library uses the names strdup and strdupl rather than strcpym and strcpyml so it will use the same names as the C standard library.

Strcpy copies the string pointed at by es:di to the memory locations beginning at the address in dx:si. There is no error checking; you must ensure that there is sufficient free space at location dx:si before calling strcpy. Strcpy returns with es:di pointing at the destination string (that is, the original dx:si value). Strcpyl works in a similar fashion, except the source string follows the call.

Strdup duplicates the string which es:di points at and returns a pointer to the new string on the heap. Strdupl works in a similar fashion, except the string follows the call. As usual, the carry flag is set if there is a memory allocation error when using strdup or strdupl. Examples:

```
String1         byte       "Copy this string",0
String2         byte       32 dup (0)
String3         byte       32 dup (0)
StrVar1         dword      0
StrVar2         dword      0
                  .
                  .
                  .
                lesi     String1      ;See strcat for lesi definition.
                ldxi     String2      ;See strcat for ldxi definition.
                strcpy

                ldxi     String3
                strcpyl
                byte     "This string, too!",0

                lesi     String1
                strdup
                jc       error                          ;If insufficient mem.
                mov      word ptr StrVar1, di           ;Save away ptr to
                mov      word ptr StrVar1+2, es         ; string.

                strdupl
                jc       error
                byte     "Also, this string",0
                mov      word ptr StrVar2, di
                mov      word ptr StrVar2+2, es

                printf
                byte     "strcpy: %s\n"
                byte     "strcpyl: %s\n"
                byte     "strdup: %^s\n"
                byte     "strdupl: %^s\n",0
                dword    String2, String3, StrVar1, StrVar2
```

15.4.6 Strdel, Strdelm

Strdel and strdelm delete characters from a string. Strdel deletes the specified characters within the string, strdelm creates a new copy of the source string without the specified characters. On entry, es:di points at the string to manipulate, cx contains the index into the string where the deletion is to start, and ax contains the number of characters to delete from the string. On return, es:di points at the new string (which is on the heap if you call strdelm). For strdelm only, if the carry flag is set on return, there was a memory allocation error. As with all UCR StdLib string routines, the index values for the string are zero-based. That is, zero is the index of the first character in the source string. Example:

```
String1         byte     "Hello there, how are you?",0
                  .
                  .
                  .
                lesi     String1      ;See strcat for lesi definition.
                mov      cx, 5        ;Start at position five (" there")
                mov      ax, 6        ;Delete six characters.
                strdelm               ;Create a new string.
                jc       error        ;If insufficient memory.
                print
                byte     "New string:",0
                puts
```

```
                putcr

                lesi    String1
                mov     ax, 11
                mov     cx, 13
                strdel
                printf
                byte    "Modified string: %s\n",0
                dword   String1
```

This code prints the following:

```
New string: Hello, how are you?
Modified string: Hello there
```

15.4.7 Strins, Strinsl, Strinsm, Strinsml

The strins(xx) functions insert one string within another. For all four routines es:di points at the source string into you want to insert another string. Cx contains the insertion point (0..length of source string). For strins and strinsm, dx:si points at the string you wish to insert. For strinsl and strinsml, the string to insert appears as a literal constant in the code stream. Strins and strinsl insert the second string directly into the string pointed at by es:di. Strinsm and strinsml make a copy of the source string and insert the second string into that copy. They return a pointer to the new string in es:di. If there is a memory allocation error then strinsm/strinsml sets the carry flag on return. For strins and strinsl, the first string must have sufficient storage allocated to hold the new string. Examples:

```
InsertInMe      byte    "Insert >< Here",0
                byte    16 dup (0)
InsertStr       byte    "insert this",0
StrPtr1         dword   0
StrPtr2         dword   0
                  .
                  .
                  .
                lesi    InsertInMe   ;See strcat for lesi definition.
                ldxi    InsertStr    ;See strcat for ldxi definition.
                mov     cx, 8        ;Insert before "<"
                strinsm
                mov     word ptr StrPtr1, di
                mov     word ptr StrPtr1+2, es

                lesi    InsertInMe
                mov     cx, 8
                strinsml
                byte    "insert that",0
                mov     word ptr StrPtr2, di
                mov     word ptr StrPtr2+2, es

                lesi    InsertInMe
                mov     cx, 8
                strinsl
                byte    " ",0        ;Two spaces

                lesi    InsertInMe
                ldxi    InsertStr
                mov     cx, 9        ;In front of first space from above.
                strins

                printf
                byte    "First string: %^s\n"
                byte    "Second string: %^s\n"
                byte    "Third string: %s\n",0
                dword   StrPtr1, StrPtr2, InsertInMe
```

Note that the strins and strinsl operations above both insert strings into the same destination string. The output from the above code is

```
First string: Insert >insert this< here
Second string: Insert >insert that< here
Third string: Insert > insert this < here
```

15.4.8 Strlen

Strlen computes the length of the string pointed at by es:di. It returns the number of characters up to, but not including, the zero terminating byte. It returns this length in the cx register. Example:

```
GetLen          byte    "This string is 33 characters long",0
                  .
                  .
                  .
                lesi    GetLen      ;See strcat for lesi definition.
                strlen
                print
                byte    "The string is ",0
                mov     ax, cx      ;Puti needs the length in AX!
                puti
                print
                byte    " characters long",cr,lf,0
```

15.4.9 Strlwr, Strlwrm, Strupr, Struprm

Strlwr and Strlwrm convert any upper case characters in a string to lower case. Strupr and Struprm convert any lower case characters in a string to upper case. These routines do not affect any other characters present in the string. For all four routines, es:di points at the source string to convert. Strlwr and strupr modify the characters directly in that string. Strlwrm and struprm make a copy of the string to the heap and then convert the characters in the new string. They also return a pointer to this new string in es:di. As usual for UCR StdLib routines, strlwrm and struprm return the carry flag set if there is a memory allocation error. Examples:

```
String1         byte    "This string has lower case.",0
String2         byte    "THIS STRING has Upper Case.",0
StrPtr1         dword   0
StrPtr2         dword   0
                  .
                  .
                  .
                lesi    String1     ;See strcat for lesi definition.
                struprm             ;Convert lower case to upper case.
                jc      error
                mov     word ptr StrPtr1, di
                mov     word ptr StrPtr1+2, es

                lesi    String2
                strlwrm             ;Convert upper case to lower case.
                jc      error
                mov     word ptr StrPtr2, di
                mov     word ptr StrPtr2+2, es

                lesi    String1
                strlwr              ;Convert to lower case, in place.

                lesi    String2
                strupr              ;Convert to upper case, in place.

                printf
                byte    "struprm: %^s\n"
                byte    "strlwrm: %^s\n"
                byte    "strlwr: %s\n"
                byte    "strupr: %s\n",0
                dword   StrPtr1, StrPtr2, String1, String2
```

The above code fragment prints the following:

```
struprm: THIS STRING HAS LOWER CASE
strlwrm: this string has upper case
strlwr: this string has lower case
strupr: THIS STRING HAS UPPER CASE
```

15.4.10 Strrev, Strrevm

These two routines reverse the characters in a string. For example, if you pass strrev the string "ABCDEF" it will convert that string to "FEDCBA". As you'd expect by now, the strrev routine reverse the string whose address you pass in es:di; strrevm first makes a copy of the string on the heap and reverses those characters leaving the original string unchanged. Of course strrevm will return the carry flag set if there was a memory allocation error. Example:

```
Palindrome      byte    "radar",0
NotPaldrm       byte    "x + y - z",0
StrPtr1         dword   0
                  .
                  .
                  .
                lesi    Palindrome  ;See strcat for lesi definition.
                strrevm
                jc      error
                mov     word ptr StrPtr1, di
                mov     word ptr StrPtr1+2, es

                lesi    NotPaldrm
                strrev

                printf
                byte    "First string: %^s\n"
                byte    "Second string: %s\n",0
                dword   StrPtr1, NotPaldrm
```

The above code produces the following output:

```
First string: radar
Second string: z - y + x
```

15.4.11 Strset, Strsetm

Strset and strsetm replicate a single character through a string. Their behavior, however, is not quite the same. In particular, while strsetm is quite similar to the *repeat* function (see "Repeat" on page 840), strset is not. Both routines expect a single character value in the al register. They will replicate this character throughout some string. Strsetm also requires a count in the cx register. It creates a string on the heap consisting of cx characters and returns a pointer to this string in es:di (assuming no memory allocation error). Strset, on the other hand, expects you to pass it the address of an existing string in es:di. It will replace each character in that string with the character in al. Note that you do not specify a length when using the strset function, strset uses the length of the existing string. Example:

```
String1         byte    "Hello there",0
                  .
                  .
                  .
                lesi    String1     ;See strcat for lesi definition.
                mov     al, '*'
                strset

                mov     cx, 8
                mov     al, '#'
                strsetm

                print
```

```
                byte    "String2: ",0
                puts
                printf
                byte    "\nString1: %s\n",0
                dword   String1
```

The above code produces the output:

```
String2: ########
String1: **********
```

15.4.12 Strspan, Strspanl, Strcspan, Strcspanl

These four routines search through a string for a character which is either in some specified character set (strspan, strspanl) or not a member of some character set (strcspan, strcspanl). These routines appear in the UCR Standard Library only because of their appearance in the C standard library. You should rarely use these routines. The UCR Standard Library includes some other routines for manipulating character sets and performing character matching operations. Nonetheless, these routines are somewhat useful on occasion and are worth a mention here.

These routines expect you to pass them the addresses of two strings: a source string and a character set string. They expect the address of the source string in es:di. Strspan and strcspan want the address of the character set string in dx:si; the character set string follows the call with strspanl and strcspanl. On return, cx contains an index into the string, defined as follows:

strspan, strspanl: Index of first character in source found in the character set.

strcspan, strcspanl: Index of first character in source *not* found in the character set.

If all the characters are in the set (or are not in the set) then cx contains the index into the string of the zero terminating byte.

Example:

```
Source          byte    "ABCDEFG 0123456",0
Set1            byte    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0
Set2            byte    "0123456789",0
Index1          word    ?
Index2          word    ?
Index3          word    ?
Index4          word    ?
                  .
                  .
                  .
                lesi    Source      ;See strcat for lesi definition.
                ldxi    Set1        ;See strcat for ldxi definition.
                strspan             ;Search for first ALPHA char.
                mov     Index1, cx  ;Index of first alphabetic char.

                lesi    Source
                lesi    Set2
                strspan             ;Search for first numeric char.
                mov     Index2, cx

                lesi    Source
                strcspanl
                byte    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0
                mov     Index3, cx

                lesi    Set2
                strcspnl
                byte    "0123456789",0
                mov     Index4, cx

                printf
                byte    "First alpha char in Source is at offset %d\n"
                byte    "First numeric char is at offset %d\n"
```

```
                byte    "First non-alpha in Source is at offset %d\n"
                byte    "First non-numeric in Set2 is at offset %d\n",0
                dword   Index1, Index2, Index3, Index4
```

This code outputs the following:

```
First alpha char in Source is at offset 0
First numeric char is at offset 8
First non-alpha in Source is at offset 7
First non-numeric in Set2 is at offset 10
```

15.4.13 Strstr, Strstrl

Strstr searches for the first occurrence of one string within another. es:di contains the address of the string in which you want to search for a second string. dx:si contains the address of the second string for the strstr routine; for strstrl the search second string immediately follows the call in the code stream.

On return from strstr or strstrl, the carry flag will be set if the second string is not present in the source string. If the carry flag is clear, then the second string is present in the source string and cx will contain the (zero-based) index where the second string was found. Example:

```
SourceStr       byte    "Search for 'this' in this string",0
SearchStr       byte    "this",0
                  .
                  .
                  .
                lesi    SourceStr   ;See strcat for lesi definition.
                ldxi    SearchStr   ;See strcat for ldxi definition.
                strstr
                jc      NotPresent
                print
                byte    "Found string at offset ",0
                mov     ax, cx      ;Need offset in AX for puti
                puti
                putcr

                lesi    SourceStr
                strstrl
                byte    "for",0
                jc      NotPresent
                print
                byte    "Found 'for' at offset ",0
                mov     ax, cx
                puti
                putcr
NotPresent:
```

The above code prints the following:

```
Found string at offset 12
Found 'for' at offset 7
```

15.4.14 Strtrim, Strtrimm

These two routines are quite similar to strbdel and strbdelm. Rather than removing leading spaces, however, they trim off any trailing spaces from a string. Strtrim trims off any trailing spaces directly on the specified string in memory. Strtrimm first copies the source string and then trims and space off the copy. Both routines expect you to pass the address of the source string in es:di. Strtrimm returns a pointer to the new string (if it could allocate it) in es:di. It also returns the carry set or clear to denote error / no error. Example:

```
String1         byte    "Spaces at the end      ",0
String2         byte    "    Spaces on both sides     ",0
StrPtr1         dword   0
StrPtr2         dword   0
                  .
                  .
                  .

; TrimSpcs trims the spaces off both ends of a string.
; Note that it is a little more efficient to perform the
; strbdel first, then the strtrim. This routine creates
; the new string on the heap and returns a pointer to this
; string in ES:DI.

TrimSpcs        proc
                strbdelm
                jc      BadAlloc    ;Just return if error.
                strtrim
                clc
BadAlloc:       ret
TrimSpcs        endp
                          .
                          .
                          .
                lesi    String1     ;See strcat for lesi definition.
                strtrimm
                jc      error
                mov     word ptr StrPtr1, di
                mov     word ptr StrPtr1+2, es

                lesi    String2
                call    TrimSpcs
                jc      error
                mov     word ptr StrPtr2, di
                mov     word ptr StrPtr2+2, es

                printf
                byte    "First string: '%s'\n"
                byte    "Second string: '%s'\n",0
                dword   StrPtr1, StrPtr2
```

This code fragment outputs the following:

```
First string: 'Spaces at the end'
Second string: 'Spaces on both sides'
```

15.4.15 Other String Routines in the UCR Standard Library

In addition to the "strxxx" routines listed in this section, there are many additional string routines available in the UCR Standard Library. Routines to convert from numeric types (integer, hex, real, etc.) to a string or vice versa, pattern matching and character set routines, and many other conversion and string utilities. The routines described in this chapter are those whose definitions appear in the "strings.a" header file and are specifically targeted towards generic string manipulation. For more details on the other string routines, consult the UCR Standard Library reference section in the appendices.

15.5 The Character Set Routines in the UCR Standard Library

The UCR Standard Library provides an extensive collection of character set routines. These routines let you create sets, clear sets (set them to the empty set), add and remove one or more items, test for set membership, copy sets, compute the union, intersection, or difference, and extract items from a set. Although intended to manipulate sets of characters, you can use the StdLib character set routines to manipulate any set with 256 or fewer possible items.

The first unusual thing to note about the StdLib's sets is their storage format. A 256-bit array would normally consumes 32 consecutive bytes. For performance reasons, the UCR Standard Library's set format packs eight separate sets into 272 bytes (256 bytes for the eight sets plus 16 bytes overhead). To declare set variables in your data segment you should use the set macro. This macro takes the form:

```
set     SetName1, SetName2, ..., SetName8
```

SetName1..SetName8 represent the names of up to eight set variables. You may have fewer than eight names in the operand field, but doing so will waste some bits in the set array.

The CreateSets routine provides another mechanism for creating set variables. Unlike the set macro, which you would use to create set variables in your data segment, the CreateSets routine allocates storage for up to eight sets dynamically at run time. It returns a pointer to the first set variable in es:di. The remaining seven sets follow at locations es:di+1, es:di+2, ..., es:di+7. A typical program that allocates set variables dynamically might use the following code:

```
Set0            dword   ?
Set1            dword   ?
Set2            dword   ?
Set3            dword   ?
Set4            dword   ?
Set5            dword   ?
Set6            dword   ?
Set7            dword   ?
                  .
                  .
                  .
                CreateSets
                mov     word ptr Set0+2, es
                mov     word ptr Set1+2, es
                mov     word ptr Set2+2, es
                mov     word ptr Set3+2, es
                mov     word ptr Set4+2, es
                mov     word ptr Set5+2, es
                mov     word ptr Set6+2, es
                mov     word ptr Set7+2, es

                mov     word ptr Set0, di
                inc     di
                mov     word ptr Set1, di
                inc     di
                mov     word ptr Set2, di
                inc     di
                mov     word ptr Set3, di
                inc     di
                mov     word ptr Set4, di
                inc     di
                mov     word ptr Set5, di
                inc     di
                mov     word ptr Set6, di
                inc     di
                mov     word ptr Set7, di
                inc     di
```

This code segment creates eight different sets on the heap, all empty, and stores pointers to them in the appropriate pointer variables.

The SHELL.ASM file provides a commented-out line of code in the data segment that includes the file STDSETS.A. This include file provides the bit definitions for eight commonly used character sets. They are alpha (upper and lower case alphabetics), lower (lower case alphabetics), upper (upper case alphabetics), digits ("0"..”9”), xdigits ("0"..”9”, “A”..”F”, and “a”..”f”), alphanum (upper and lower case alphabetics plus the digits), whitespace (space, tab, carriage return, and line feed), and delimiters (whitespace plus commas, semicolons, less than, greater than, and vertical bar). If you would like to use these standard character sets in your program, you need to remove the semicolon from the beginning of the include statement in the SHELL.ASM file.

The UCR Standard Library provides 16 character set routines: CreateSets, EmptySet, RangeSet, AddStr, AddStrl, RmvStr, RmvStrl, AddChar, RmvChar, Member, CopySet, SetUnion, SetIntersect, SetDifference, NextItem, and RmvItem. All of these routines except CreateSets require a pointer to a character set variable in the es:di registers. Specific routines may require other parameters as well.

The EmptySet routine clears all the bits in a set producing the empty set. This routine requires the address of the set variable in the es:di. The following example clears the set pointed at by Set1:

```
                les     di, Set1
                EmptySet
```

RangeSet unions in a range of values into the set variable pointed at by es:di. The al register contains the lower bound of the range of items, ah contains the upper bound. Note that al must be less than or equal to ah. The following example constructs the set of all control characters (ASCII codes one through 31, the null character [ASCII code zero] is not allowed in sets):

```
                les     di, CtrlCharSet         ;Ptr to ctrl char set.
                mov     al, 1
                mov     ah, 31
                RangeSet
```

AddStr and AddStrl add all the characters in a zero terminated string to a character set. For AddStr, the dx:si register pair points at the zero terminated string. For AddStrl, the zero terminated string follows the call to AddStrl in the code stream. These routines union each character of the specified string into the set. The following examples add the digits and some special characters into the FPDigits set:

```
Digits          byte    "0123456789",0
                set     FPDigitsSet
FPDigits        dword   FPDigitsSet
                  .
                  .
                  .
                ldxi    Digits      ;Loads DX:SI with adrs of Digits.
                les     di, FPDigits
                AddStr
                  .
                  .
                  .
                les     di, FPDigits
                AddStrL
                byte    "Ee.+-",0
```

RmvStr and RmvStrl *remove* characters from a set. You supply the characters in a zero terminated string. For RmvStr, dx:si points at the string of characters to remove from the string. For RmvStrl, the zero terminated string follows the call. The following example uses RmvStrl to remove the special symbols from FPDigits above:

```
                les     di, FPDigits
                RmvStrl
                byte    "Ee.+-",0
```

The AddChar and RmvChar routines let you add or remove individual characters. As usual, es:di points at the set; the al register contains the character you wish to add to the set or remove from the set. The following example adds a space to the set FPDigits and removes the "," character (if present):

```
                les     di, FPDigits
                mov     al, ' '
                AddChar
                  .
                  .
                  .
                les     di, FPDigits
                mov     al, ','
                RmvChar
```

The Member function checks to see if a character is in a set. On entry, es:di must point at the set and al must contain the character to check. On exit, the zero flag is set if the character is a member of the set, the zero flag will be clear if the character is not in the set. The following example reads characters from the keyboard until the user presses a key that is not a whitespace character:

```
SkipWS:         get                     ;Read char from user into AL.
                lesi    WhiteSpace      ;Address of WS set into es:di.
                member
                je      SkipWS
```

The CopySet, SetUnion, SetIntersect, and SetDifference routines all operate on two sets of characters. The es:di register points at the destination character set, the dx:si register pair points at a source character set. CopySet copies the bits from the source set to the destination set, replacing the original bits in the destination set. SetUnion computes the union of the two sets and stores the result into the destination set. SetIntersect computes the set intersection and stores the result into the destination set. Finally, the SetDifference routine computes DestSet := DestSet - SrcSet.

The NextItem and RmvItem routines let you extract elements from a set. NextItem returns in al the ASCII code of the first character it finds in a set. RmvItem does the same thing except it also removes the character from the set. These routines return zero in al if the set is empty (StdLib sets cannot contain the NULL character). You can use the RmvItem routine to build a rudimentary iterator for a character set.

The UCR Standard Library's character set routines are very powerful. With them, you can easily manipulate character string data, especially when searching for different patterns within a string. We will consider this routines again when we study pattern matching later in this text (see "Pattern Matching" on page 883).

15.6 Using the String Instructions on Other Data Types

The string instructions work with other data types besides character strings. You can use the string instructions to copy whole arrays from one variable to another, to initialize large data structures to a single value, or to compare entire data structures for equality or inequality. Anytime you're dealing with data structures containing several bytes, you may be able to use the string instructions.

15.6.1 Multi-precision Integer Strings

The cmps instruction is useful for comparing (very) large integer values. Unlike character strings, we cannot compare integers with cmps from the L.O. byte through the H.O. byte. Instead, we must compare them from the H.O. byte down to the L.O. byte. The following code compares two 12-byte integers:

```
                lea     di, integer1+10
                lea     si, integer2+10
                mov     cx, 6
                std
    repe        cmpsw
```

After the execution of the cmpsw instruction, the flags will contain the result of the comparison.

You can easily assign one long integer string to another using the movs instruction. Nothing tricky here, just load up the si, di, and cx registers and have at it. You must do other operations, including arithmetic and logical operations, using the extended precision methods described in the chapter on arithmetic operations.

15.6.2 Dealing with Whole Arrays and Records

The only operations that apply, in general, to all array and record structures are assignment and comparison (for equality/inequality only). You can use the movs and cmps instructions for these operations.

Operations such as scalar addition, transposition, etc., may be easily synthesized using the lods and stos instructions. The following code shows how you can easily add the value 20 to each element of the integer array A:

```
                lea     si, A
                mov     di, si
                mov     cx, SizeOfA
                cld
AddLoop:        lodsw
                add     ax, 20
                stosw
                loop    AddLoop
```

You can implement other operations in a similar fashion.

15.7 Sample Programs

In this section there are three sample programs. The first searches through a file for a particular string and displays the line numbers of any lines containing that string. This program demonstrates the use of the strstr function (among other things). The second program is a demo program that uses several of the string functions available in the UCR Standard Library's string package. The third program demonstrates how to use the 80x86 cmps instruction to compare the data in two files. These programs (find.asm, strdemo.asm, and fcmp.asm) are available on the companion CD-ROM.

15.7.1 Find.asm

```
; Find.asm
;
; This program opens a file specified on the command line and searches for
; a string (also specified on the command line).
;
; Program Usage:
;
;       find "string" filename


                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

wp              textequ   <word ptr>

dseg            segment   para public 'data'

StrPtr          dword     ?
FileName        dword     ?
LineCnt         dword     ?

FVar            filevar   {}

InputLine       byte      1024 dup (?)
dseg            ends
```

```
cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


; Readln-       This procedure reads a line of text from the input
;               file and buffers it up in the "InputLine" array.

ReadLn          proc
                push     es
                push     ax
                push     di
                push     bx

                lesi     FVar         ;Read from our file.
                mov      bx, 0        ;Index into InputLine.
ReadLp:         fgetc                 ;Get next char from file.
                jc       EndRead      ;Quit on EOF

                cmp      al, cr       ;Ignore carriage returns.
                je       ReadLp
                cmp      al, lf       ;End of line on line feed.
                je       EndRead

                mov      InputLine[bx], al
                inc      bx
                jmp      ReadLp

; If we hit the end of a line or the end of the file,
; zero-terminate the string.

EndRead:        mov      InputLine[bx], 0
                pop      bx
                pop      di
                pop      ax
                pop      es
                ret
ReadLn          endp


; The following main program extracts the search string and the
; filename from the command line, opens the file, and then searches
; for the string in that file.

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

                argc
                cmp      cx, 2
                je       GoodArgs
                print
                byte     "Usage: find 'string' filename",cr,lf,0
                jmp      Quit

GoodArgs:       mov      ax, 1        ;Get the string to search for
                argv                  ; off the command line.
                mov      wp StrPtr, di
                mov      wp StrPtr+2, es

                mov      ax, 2        ;Get the filename from the
                argv                  ; command line.
                mov      wp Filename, di
                mov      wp Filename+2, es

; Open the input file for reading

                mov      ax, 0        ;Open for read.
                mov      si, wp FileName
```

```
                mov     dx, wp FileName+2
                lesi    Fvar
                fopen
                jc      BadOpen

; Okay, start searching for the string in the file.

                mov     wp LineCnt, 0
                mov     wp LineCnt+2, 0
SearchLp:       call    ReadLn
                jc      AtEOF


; Bump the line number up by one.  Note that this is 8086 code
; so we have to use extended precision arithmetic to do a 32-bit
; add.  LineCnt is a 32-bit variable because some files have more
; that 65,536 lines.

                add     wp LineCnt, 1
                adc     wp LineCnt+2, 0

; Search for the user-specified string on the current line.

                lesi    InputLine
                mov     dx, wp StrPtr+2
                mov     si, wp StrPtr
                strstr
                jc      SearchLp;Jump if not found.

; Print an appropriate message if we found the string.

                printf
                byte    "Found '%^s' at line %ld\n",0
                dword   StrPtr, LineCnt
                jmp     SearchLp

; Close the file when we're done.

AtEOF:          lesi    FVar
                fclose
                jmp     Quit

BadOpen:        printf
                byte    "Error attempting to open %^s\n",cr,lf,0
                dword   FileName


Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

15.7.2 StrDemo.asm

This short demo program just shows off how to use several of the string routines found in the UCR Standard Library strings package.

```
; StrDemo.asm-   Demonstration of some of the various UCR Standard Library
;               string routines.
```

```
                include    stdlib.a
                includelib stdlib.lib

dseg            segment  para public 'data'

MemAvail        word     ?
String          byte     256 dup (0)

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


Main            proc
                mov      ax, seg dseg ;Set up the segment registers
                mov      ds, ax
                mov      es, ax

                MemInit
                mov      MemAvail, cx
                printf
                byte     "There are %x paragraphs of memory available."
                byte     cr,lf,lf,0
                dword    MemAvail

; Demonstration of StrTrim:

                print
                byte     "Testing strtrim on 'Hello there   '",cr,lf,0
                strdupl
HelloThere1     byte     "Hello there   ",0
                strtrim
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free

;Demonstration of StrTrimm:

                print
                byte     "Testing strtrimm on 'Hello there   '",cr,lf,0
                lesi     HelloThere1
                strtrimm
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free

; Demonstration of StrBdel

                print
                byte     "Testing strbdel on '  Hello there   '",cr,lf,0
                strdupl
HelloThere3     byte     "  Hello there   ",0
                strbdel
                mov      al, "'"
                putc
                puts
                putc
                putcr
                free
```

```
; Demonstration of StrBdelm

                print
                byte    "Testing strbdelm on '  Hello there   '",cr,lf,0
                lesi    HelloThere3
                strbdelm
                mov     al, "'"
                putc
                puts
                putc
                putcr
                free


; Demonstrate StrCpyl:

                ldxi    string
                strcpyl
                byte    "Copy this string to the 'String' variable",0

                printf
                byte    "STRING = '%s'",cr,lf,0
                dword   String

; Demonstrate StrCatl:

                lesi    String
                strcatl
                byte    ". Put at end of 'String'",0

                printf
                byte    "STRING = ",'"%s"',cr,lf,0
                dword   String

; Demonstrate StrChr:

                lesi    String
                mov     al, "'"
                strchr

                print
                byte    "StrChr: First occurrence of ", '"', "'"
                byte    '" found at position ',0
                mov     ax, cx
                puti
                putcr

; Demonstrate StrStrl:

                lesi    String
                strstrl
                byte    "String",0

                print
                byte    'StrStr: First occurrence of "String" found at ‘
                byte    ‘position ',0

                mov     ax, cx
                puti
                putcr

; Demo of StrSet

                lesi    String
                mov     al, '*'
                strset

                printf
                byte    "Strset:  '%s'",cr,lf,0
                dword   String
```

```
; Demo of strlen

                lesi    String
                strlen

                print
                byte    "String length = ",0
                puti
                putcr

Quit:           mov     ah, 4ch
                int     21h
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       256 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

15.7.3 Fcmp.asm

This is a file comparison program. It demonstrates the use of the 80x86 cmps instruction (as well as blocked I/O under DOS).

```
; FCMP.ASM-     A file comparison program that demonstrates the use
;               of the 80x86 string instructions.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'

Name1           dword    ?           ;Ptr to filename #1
Name2           dword    ?           ;Ptr to filename #2
Handle1         word     ?           ;File handle for file #1
Handle2         word     ?           ;File handle for file #2
LineCnt         word     0           ;# of lines in the file.

Buffer1         byte     256 dup (0)  ;Block of data from file 1
Buffer2         byte     256 dup (0)  ;Block of data from file 2

dseg            ends

wp              equ      <word ptr>

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Error- Prints a DOS error message depending upon the error type.

Error           proc     near
                cmp      ax, 2
                jne      NotFNF
                print
                byte     "File not found",0
                jmp      ErrorDone

NotFNF:         cmp      ax, 4
                jne      NotTMF
```

```
                        print
                        byte    "Too many open files",0
                        jmp     ErrorDone

NotTMF:                 cmp     ax, 5
                        jne     NotAD
                        print
                        byte    "Access denied",0
                        jmp     ErrorDone

NotAD:                  cmp     ax, 12
                        jne     NotIA
                        print
                        byte    "Invalid access",0
                        jmp     ErrorDone

NotIA:
ErrorDone:              putcr
                        ret
Error                   endp

; Okay, here's the main program.  It opens two files, compares them, and
; complains if they're different.

Main                    proc
                        mov     ax, seg dseg            ;Set up the segment registers
                        mov     ds, ax
                        mov     es, ax
                        meminit

; File comparison routine.  First, open the two source files.

                        argc
                        cmp     cx, 2                   ;Do we have two filenames?
                        je      GotTwoNames
                        print
                        byte    "Usage: fcmp file1 file2",cr,lf,0
                        jmp     Quit

GotTwoNames:            mov     ax, 1                   ;Get first file name
                        argv
                        mov     wp Name1, di
                        mov     wp Name1+2, es

; Open the files by calling DOS.

                        mov     ax, 3d00h               ;Open for reading
                        lds     dx, Name1
                        int     21h
                        jnc     GoodOpen1
                        printf
                        byte    "Error opening %^s:",0
                        dword   Name1
                        call    Error
                        jmp     Quit

GoodOpen1:              mov     dx, dseg
                        mov     ds, dx
                        mov     Handle1, ax

                        mov     ax, 2                   ;Get second file name
                        argv
                        mov     wp Name2, di
                        mov     wp Name2+2, es

                        mov     ax, 3d00h               ;Open for reading
                        lds     dx, Name2
                        int     21h
                        jnc     GoodOpen2
                        printf
```

```
                byte    "Error opening %^s:",0
                dword   Name2
                call    Error
                jmp     Quit

GoodOpen2:      mov     dx, dseg
                mov     ds, dx
                mov     Handle2, ax

; Read the data from the files using blocked I/O
; and compare it.

                mov     LineCnt, 1
CmpLoop:        mov     bx, Handle1         ;Read 256 bytes from
                mov     cx, 256             ; the first file into
                lea     dx, Buffer1         ; Buffer1.
                mov     ah, 3fh
                int     21h
                jc      FileError
                cmp     ax, 256             ;Leave if at EOF.
                jne     EndOfFile

                mov     bx, Handle2         ;Read 256 bytes from
                mov     cx, 256             ; the second file into
                lea     dx, Buffer2         ; Buffer2
                mov     ah, 3fh
                int     21h
                jc      FileError
                cmp     ax, 256             ;If we didn't read 256 bytes,
                jne     BadLen              ; the files are different.

; Okay, we've just read 256 bytes from each file, compare the buffers
; to see if the data is the same in both files.

                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                mov     cx, 256
                lea     di, Buffer1
                lea     si, Buffer2
                cld
        repe    cmpsb
                jne     BadCmp
                jmp     CmpLoop


FileError:      print
                byte    "Error reading files: ",0
                call    Error
                jmp     Quit


BadLen:         print
                byte    "File lengths were different",cr,lf,0

BadCmp:         print
                byte    7,"Files were not equal",cr,lf,0

                mov     ax, 4c01h   ;Exit with error.
                int     21h


; If we reach the end of the first file, compare any remaining bytes
; in that first file against the remaining bytes in the second file.

EndOfFile:      push    ax          ;Save final length.
                mov     bx, Handle2
                mov     cx, 256
                lea     dx, Buffer2
                mov     ah, 3fh
```

```
                int     21h
                jc      BadCmp

                pop     bx          ;Retrieve file1's length.
                cmp     ax, bx      ;See if file2 matches it.
                jne     BadLen

                mov     cx, ax      ;Compare the remaining
                mov     ax, dseg    ; bytes down here.
                mov     ds, ax
                mov     es, ax
                lea     di, Buffer2
                lea     si, Buffer1
        repe    cmpsb
                jne     BadCmp

Quit:           mov     ax, 4c00h   ;Set Exit code to okay.
                int     21h
Main            endp
cseg            ends

; Allocate a reasonable amount of space for the stack (2k).

sseg            segment  para stack 'stack'
stk             byte     256 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup (?)
zzzzzzseg       ends
                end      Main
```

15.8 Laboratory Exercises

These exercises use the Ex15_1.asm, Ex15_2.asm, Ex15_3.asm, and Ex15_4.asm files found on the companion CD-ROM. In this set of laboratory exercises you will be measuring the performance of the 80x86 movs instructions and the (hopefully) minor performance differences between length prefixed string operations and zero terminated string operations.

15.8.1 MOVS Performance Exercise #1

The movsb, movsw, and movsd instructions operate at different speeds, even when moving around the same number of bytes. In general, the movsw instruction is twice as fast as movsb when moving the same number of bytes. Likewise, movsd is about twice as fast as movsw (and about four times as fast as movsb) when moving the same number of bytes. Ex15_1.asm is a short program that demonstrates this fact. This program consists of three sections that copy 2048 bytes from one buffer to another 100,000 times. The three sections repeat this operation using the movsb, movsw, and movsd instructions. Run this program and time each phase. **For your lab report:** present the timings on your machine. Be sure to list processor type and clock frequency in your lab report. Discuss why the timings are different between the three phases of this program. Explain the difficulty with using the movsd (versus movsw or movsb) instruction in any program on an 80386 or later processor. Why is it not a general replacement for movsb, for example? How can you get around this problem?

```
; EX15_1.asm
;
; This program demonstrates the proper use of the 80x86 string instructions.

                .386
                option      segment:use16
```

```
                include     stdlib.a
                includelib  stdlib.lib


dseg            segment  para public 'data'

Buffer1         byte     2048 dup (0)
Buffer2         byte     2048 dup (0)

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit

; Demo of the movsb, movsw, and movsd instructions

                print
                byte     "The following code moves a block of 2,048 bytes "
                byte     "around 100,000 times.",cr,lf
                byte     "The first phase does this using the movsb "
                byte     "instruction; the second",cr,lf
                byte     "phase does this using the movsw instruction; "
                byte     "the third phase does",cr,lf
                byte     "this using the movsd instruction.",cr,lf,lf,lf
                byte     "Press any key to begin phase one:",0

                getc
                putcr

                mov      edx, 100000

movsbLp:        lea      si, Buffer1
                lea      di, Buffer2
                cld
                mov      cx, 2048
       rep      movsb
                dec      edx
                jnz      movsbLp

                print
                byte     cr,lf
                byte     "Phase one complete",cr,lf,lf
                byte     "Press any key to begin phase two:",0

                getc
                putcr

                mov      edx, 100000

movswLp:        lea      si, Buffer1
                lea      di, Buffer2
                cld
                mov      cx, 1024
       rep      movsw
                dec      edx
                jnz      movswLp

                print
                byte     cr,lf
                byte     "Phase two complete",cr,lf,lf
                byte     "Press any key to begin phase three:",0

                getc
```

```
                putcr

                mov     edx, 100000

movsdLp:        lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 512
       rep      movsd
                dec     edx
                jnz     movsdLp


Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

15.8.2 MOVS Performance Exercise #2

In this exercise you will once again time the computer moving around blocks of 2,048 bytes. Like Ex15_1.asm in the previous exercise, Ex15_2.asm contains three phases; the first phase moves data using the movsb instruction; the second phase moves the data around using the lodsb and stosb instructions; the third phase uses a loop with simple mov instructions. Run this program and time the three phases. **For your lab report:** include the timings and a description of your machine (CPU, clock speed, etc.). Discuss the timings and explain the results (consult Appendix D as necessary).

```
; EX15_2.asm
;
; This program compares the performance of the MOVS instruction against
; a manual block move operation.  It also compares MOVS against a LODS/STOS
; loop.

                .386
                option     segment:use16

                include    stdlib.a
                includelib stdlib.lib


dseg            segment  para public 'data'

Buffer1         byte     2048 dup (0)
Buffer2         byte     2048 dup (0)

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit
```

```
; MOVSB version done here:

                print
                byte    "The following code moves a block of 2,048 bytes "
                byte    "around 100,000 times.",cr,lf
                byte    "The first phase does this using the movsb "
                byte    "instruction; the second",cr,lf
                byte    "phase does this using the lods/stos instructions; "
                byte    "the third phase does",cr,lf
                byte    "this using a loop with MOV “
                byte    “instructions.",cr,lf,lf,lf
                byte    "Press any key to begin phase one:",0

                getc
                putcr

                mov     edx, 100000

movsbLp:        lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 2048
        rep     movsb
                dec     edx
                jnz     movsbLp

                print
                byte    cr,lf
                byte    "Phase one complete",cr,lf,lf
                byte    "Press any key to begin phase two:",0

                getc
                putcr

                mov     edx, 100000

LodsStosLp:     lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 2048
lodsstoslp2:    lodsb
                stosb
                loop    LodsStosLp2
                dec     edx
                jnz     LodsStosLp

                print
                byte    cr,lf
                byte    "Phase two complete",cr,lf,lf
                byte    "Press any key to begin phase three:",0

                getc
                putcr

                mov     edx, 100000

MovLp:          lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 2048
MovLp2:         mov     al, ds:[si]
                mov     es:[di], al
                inc     si
                inc     di
                loop    MovLp2
                dec     edx
                jnz     MovLp
```

```
Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

15.8.3 Memory Performance Exercise

In the previous two exercises, the programs accessed a maximum of 4K of data. Since most modern on-chip CPU caches are at least this big, most of the activity took place directly on the CPU (which is very fast). The following exercise is a slight modification that moves the array data in such a way as to destroy cache performance. Run this program and time the results. **For your lab report:** based on what you learned about the 80x86's cache mechanism in Chapter Three, explain the performance differences.

```
; EX15_3.asm
;
; This program compares the performance of the MOVS instruction against
; a manual block move operation.  It also compares MOVS against a LODS/STOS
; loop.  This version does so in such a way as to wipe out the on-chip CPU
; cache.

                .386
                option     segment:use16

                include    stdlib.a
                includelib stdlib.lib

dseg            segment  para public 'data'

Buffer1         byte     16384 dup (0)
Buffer2         byte     16384 dup (0)

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg
                mov      ds, ax
                mov      es, ax
                meminit


; MOVSB version done here:

                print
                byte     "The following code moves a block of 16,384 bytes "
                byte     "around 12,500 times.",cr,lf
                byte     "The first phase does this using the movsb "
                byte     "instruction; the second",cr,lf
                byte     "phase does this using the lods/stos instructions; "
                byte     "the third phase does",cr,lf
                byte     "this using a loop with MOV instructions."
                byte     cr,lf,lf,lf
                byte     "Press any key to begin phase one:",0

                getc
```

```
                putcr

                mov     edx, 12500

movsbLp:        lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 16384
        rep     movsb
                dec     edx
                jnz     movsbLp

                print
                byte    cr,lf
                byte    "Phase one complete",cr,lf,lf
                byte    "Press any key to begin phase two:",0

                getc
                putcr

                mov     edx, 12500

LodsStosLp:     lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 16384
lodsstoslp2:    lodsb
                stosb
                loop    LodsStosLp2
                dec     edx
                jnz     LodsStosLp

                print
                byte    cr,lf
                byte    "Phase two complete",cr,lf,lf
                byte    "Press any key to begin phase three:",0

                getc
                putcr

                mov     edx, 12500

MovLp:          lea     si, Buffer1
                lea     di, Buffer2
                cld
                mov     cx, 16384
MovLp2:         mov     al, ds:[si]
                mov     es:[di], al
                inc     si
                inc     di
                loop    MovLp2
                dec     edx
                jnz     MovLp

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

15.8.4 The Performance of Length-Prefixed vs. Zero-Terminated Strings

The following program (Ex15_4.asm on the companion CD-ROM) executes two million string operations. During the first phase of execution, this code executes a sequence of length-prefixed string operations 1,000,000 times. During the second phase it does a comparable set of operation on zero terminated strings. Measure the execution time of each phase. **For your lab report:** report the differences in execution times and comment on the relative efficiency of length prefixed vs. zero terminated strings. Note that the relative performances of these sequences will vary depending upon the processor you use. Based on what you learned in Chapter Three and the cycle timings in Appendix D, explain some possible reasons for relative performance differences between these sequences among different processors.

```
; EX15_4.asm
;
; This program compares the performance of length prefixed strings versus
; zero terminated strings using some simple examples.
;
; Note: these routines all assume that the strings are in the data segment
;       and both ds and es already point into the data segment.

                .386
                option      segment:use16

                include     stdlib.a
                includelib  stdlib.lib

dseg            segment   para public 'data'

LStr1           byte      17,"This is a string."
LResult         byte      256 dup (?)

ZStr1           byte      "This is a string",0
ZResult         byte      256 dup (?)

dseg            ends


cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg

; LStrCpy: Copies a length prefixed string pointed at by SI to
;          the length prefixed string pointed at by DI.

LStrCpy         proc
                push      si
                push      di
                push      cx

                cld

                mov       cl, [si]      ;Get length of string.
                mov       ch, 0
                inc       cx            ;Include length byte.
       rep      movsb

                pop       cx
                pop       di
                pop       si
                ret
LStrCpy         endp

; LStrCat-      Concatenates the string pointed at by SI to the end
;               of the string pointed at by DI using length
;               prefixed strings.

LStrCat         proc
```

```
                push    si
                push    di
                push    cx

                cld

; Compute the final length of the concatenated string

                mov     cl, [di]                ;Get orig length.
                mov     ch, [si]                ;Get 2nd Length.
                add     [di], ch                ;Compute new length.

; Move SI to the first byte beyond the end of the first string.

                mov     ch, 0                   ;Zero extend orig len.
                add     di, cx                  ;Skip past str.
                inc     di                      ;Skip past length byte.

; Concatenate the second string (SI) to the end of the first string (DI)

        rep     movsb                           ;Copy 2nd to end of orig.

                pop     cx
                pop     di
                pop     si
                ret
LStrCat         endp

; LStrCmp-      String comparison using two length prefixed strings.
;               SI points at the first string, DI points at the
;               string to compare it against.

LStrCmp         proc
                push    si
                push    di
                push    cx

                cld

; When comparing the strings, we need to compare the strings
; up to the length of the shorter string.  The following code
; computes the minimum length of the two strings.

                mov     cl, [si]    ;Get the minimum of the two lengths
                mov     ch, [di]
                cmp     cl, ch
                jb      HasMin
                mov     cl, ch
HasMin:         mov     ch, 0

        repe    cmpsb               ;Compare the two strings.
                je      CmpLen
                pop     cx
                pop     di
                pop     si
                ret

; If the strings are equal through the length of the shorter string,
; we need to compare their lengths

CmpLen:         pop     cx
                pop     di
                pop     si

                mov     cl, [si]
                cmp     cl, [di]
                ret
LStrCmp         endp

; ZStrCpy- Copies the zero terminated string pointed at by SI
```

```
;          to the zero terminated string pointed at by DI.

ZStrCpy          proc
                 push     si
                 push     di
                 push     ax

ZSCLp:           mov      al, [si]
                 inc      si
                 mov      [di], al
                 inc      di
                 cmp      al, 0
                 jne      ZSCLp

                 pop      ax
                 pop      di
                 pop      si
                 ret
ZStrCpy          endp


; ZStrCat-       Concatenates the string pointed at by SI to the end
;                of the string pointed at by DI using zero terminated
;                strings.

ZStrCat          proc
                 push     si
                 push     di
                 push     cx
                 push     ax

                 cld

; Find the end of the destination string:

                 mov      cx, 0FFFFh
                 mov      al, 0        ;Look for zero byte.
       repne     scasb

; Copy the source string to the end of the destination string:

ZcatLp:          mov      al, [si]
                 inc      si
                 mov      [di], al
                 inc      di
                 cmp      al, 0
                 jne      ZCatLp

                 pop      ax
                 pop      cx
                 pop      di
                 pop      si
                 ret
ZStrCat          endp

; ZStrCmp-       Compares two zero terminated strings.
;                This is actually easier than the length
;                prefixed comparison.

ZStrCmp          proc
                 push     cx
                 push     si
                 push     di

; Compare the two strings until they are not equal
; or until we encounter a zero byte.  They are equal
; if we encounter a zero byte after comparing the
; two characters from the strings.

ZCmpLp:          mov      al, [si]
```

```
                inc     si
                cmp     al, [di]
                jne     ZCmpDone
                inc     di
                cmp     al, 0
                jne     ZCmpLp

ZCmpDone:       pop     di
                pop     si
                pop     cx
                ret
ZStrCmp         endp


Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                print
                byte    "The following code does 1,000,000 string "
                byte    "operations using",cr,lf
                byte    "length prefixed strings.  Measure the amount "
                byte    "of time this code",cr,lf
                byte    "takes to run.",cr,lf,lf
                byte    "Press any key to begin:",0

                getc
                putcr

                mov     edx, 1000000
LStrCpyLp:      lea     si, LStr1
                lea     di, LResult
                call    LStrCpy
                call    LStrCat
                call    LStrCat
                call    LStrCat
                call    LStrCpy
                call    LStrCmp
                call    LStrCat
                call    LStrCmp

                dec     edx
                jne     LStrCpyLp

                print
                byte    "The following code does 1,000,000 string "
                byte    "operations using",cr,lf
                byte    "zero terminated strings.  Measure the amount "
                byte    "of time this code",cr,lf
                byte    "takes to run.",cr,lf,lf
                byte    "Press any key to begin:",0

                getc
                putcr

                mov     edx, 1000000
ZStrCpyLp:      lea     si, ZStr1
                lea     di, ZResult
                call    ZStrCpy
                call    ZStrCat
                call    ZStrCat
                call    ZStrCat
                call    ZStrCpy
                call    ZStrCmp
                call    ZStrCat
                call    ZStrCmp

                dec     edx
```

```
                jne     ZStrCpyLp

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

15.9 Programming Projects

1) Write a SubStr function that extracts a substring from a zero terminated string. Pass a pointer to the string in ds:si, a pointer to the destination string in es:di, the starting position in the string in ax, and the length of the substring in cx. Follow all the rules given in section 15.3.1 concerning degenerate conditions.

2) Write a word *iterator* (see "Iterators" on page 663) to which you pass a string (by reference, on the stack). Each each iteration of the corresponding foreach loop should extract a word from this string, malloc sufficient storage for this string on the heap, copy that word (substring) to the malloc'd location, and return a pointer to the word. Write a main program that calls the iterator with various strings to test it.

3) Modify the *find.asm* program (see "Find.asm" on page 860) so that it searches for the desired string in several files using ambiguous filenames (i.e., wildcard characters). See "Find First File" on page 729 for details about processing filenames that contain wildcard characters. You should write a loop that processes all matching filenames and executes the find.asm core code on each filename that matches the ambiguous filename a user supplies.

4) Write a strncpy routine that behaves like strcpy except it copies a maximum of n characters (including the zero terminating byte). Pass the source string's address in es:di, the destination string's address in dx:si, and the maximum length in cx.

5) The movsb instruction may not work properly if the source and destination blocks overlap (see "The MOVS Instruction" on page 822). Write a procedure "bcopy" to which you pass the address of a source block, the address of a destination block, and a length, that will properly copy the data even if the source and destination blocks overlap. Do this by checking to see if the blocks overlap and adjusting the source pointer, destination pointer, and direction flag if necessary.

6) As you discovered in the lab experiments, the movsd instruction can move a block of data much faster than movsb or movsw can move that same block. Unfortunately, it can only move a block that contains an even multiple of four bytes. Write a "fastcopy" routine that uses the movsd instruction to copy all but the last one to three bytes of a source block to the destination block and then manually copies the remaining bytes between the blocks. Write a main program with several boundary test cases to verify correct operation. Compare the performance of your fastcopy procedure against the use of the movsb instruction.

15.10 Summary

The 80sx86 provides a powerful set of string instructions. However, these instructions are very primitive, useful mainly for manipulating blocks of bytes. They do not correspond to the string instructions one expects to find in a high level language. You can, however, use the 80x86 string instructions to synthesize those functions normally associated with HLLs. This chapter explains how to construct many of the more popular string func-

tions. Of course, it's foolish to constantly reinvent the wheel, so this chapter also describes many of the string functions available in the UCR Standard Library.

The 80x86 string instructions provide the basis for many of the string operations appearing in this chapter. Therefore, this chapter begins with a review and in-depth discussion of the 80x86 string instructions: the repeat prefixes, and the direction flag. This chapter discusses the operation of each of the string instructions and describes how you can use each of them to perform string related tasks. To see how the 80x86 string instructions operate, check out the following sections:

- "The 80x86 String Instructions" on page 819
- "How the String Instructions Operate" on page 819
- "The REP/REPE/REPZ and REPNZ/REPNE Prefixes" on page 820
- "The Direction Flag" on page 821
- "The MOVS Instruction" on page 822
- "The CMPS Instruction" on page 826
- "The SCAS Instruction" on page 828
- "The STOS Instruction" on page 828
- "The LODS Instruction" on page 829
- "Building Complex String Functions from LODS and STOS" on page 830
- "Prefixes and the String Instructions" on page 830

Although Intel calls them "string instructions" they do not actually work on the abstract data type we normally think of as a character string. The string instructions simply manipulate arrays of bytes, words, or double words. It takes a little work to get these instructions to deal with true character strings. Unfortunately, there isn't a single definition of a character string which, no doubt, is the reason there aren't any instructions specifically for character strings in the 80x86 instruction set. Two of the more popular character string types include length prefixed strings and zero terminated strings which Pascal and C use, respectively. Details on string formats appear in the following sections:

- "Character Strings" on page 831
- "Types of Strings" on page 831

Once you decide on a specific data type for you character strings, the next step is to implement various functions to process those strings. This chapter provides examples of several different string functions designed specifically for length prefixed strings. To learn about these functions and see the code that implements them, look at the following sections:

- "String Assignment" on page 832
- "String Comparison" on page 834
- "Character String Functions" on page 835
- "Substr" on page 835
- "Index" on page 838
- "Repeat" on page 840
- "Insert" on page 841
- "Delete" on page 843
- "Concatenation" on page 844

The UCR Standard Library provides a very rich set of string functions specifically designed for zero germinated strings. For a description of many of these routines, read the following sections:

- "String Functions in the UCR Standard Library" on page 845
- "StrBDel, StrBDelm" on page 846
- "Strcat, Strcatl, Strcatm, Strcatml" on page 847
- "Strchr" on page 848
- "Strcmp, Strcmpl, Stricmp, Stricmpl" on page 848
- "Strcpy, Strcpyl, Strdup, Strdupl" on page 849

- “Strdel, Strdelm” on page 850
- “Strins, Strinsl, Strinsm, Strinsml” on page 851
- “Strlen” on page 852
- “Strlwr, Strlwrm, Strupr, Struprm” on page 852
- “Strrev, Strrevm” on page 853
- “Strset, Strsetm” on page 853
- “Strspan, Strspanl, Strcspan, Strcspanl” on page 854
- “Strstr, Strstrl” on page 855
- “Strtrim, Strtrimm” on page 855
- “Other String Routines in the UCR Standard Library” on page 856

As mentioned earlier, the string instructions are quite useful for many operations beyond character string manipulation. This chapter closes with some sections describing other uses for the string instructions. See

- “Using the String Instructions on Other Data Types” on page 859
- “Multi-precision Integer Strings” on page 859
- “Dealing with Whole Arrays and Records” on page 860

The set is another common abstract data type commonly found in programs today. A set is a data structure which represent membership (or lack thereof) of some group of objects. If all objects are of the same underlying base type and there is a limited number of possible objects in the set, then we can use a *bit vector* (array of booleans) to represent the set. The bit vector implementation is very efficient for small sets. The UCR Standard Library provides several routines to manipulate character sets and other sets with a maximum of 256 members. For more details,

- “The Character Set Routines in the UCR Standard Library” on page 856

15.11 Questions

1) What are the repeat prefixes used for?

2) Which string prefixes are used with the following instructions?

   a) MOVS b) CMPS c) STOS d) SCAS

3) Why aren't the repeat prefixes normally used with the LODS instruction?

4) What happens to the SI, DI, and CX registers when the MOVSB instruction is executed (without a repeat prefix) and:

   a) the direction flag is set. b) the direction flag is clear.

5) Explain how the MOVSB and MOVSW instructions work. Describe how they affect memory and registers with and without the repeat prefix. Describe what happens when the direction flag is set and clear.

6) How do you preserve the value of the direction flag across a procedure call?

7) How can you ensure that the direction flag always contains a proper value before a string instruction without saving it inside a procedure?

8) What is the difference between the "MOVSB", "MOVSW", and "MOVS oprnd1,oprnd2" instructions?

9) Consider the following Pascal array definition:

```
a:array [0..31] of record
            a,b,c:char;
            i,j,k:integer;
          end;
```

   Assuming A[0] has been initialized to some value, explain how you can use the MOVS instruction to initialize the remaining elements of A to the same value as A[0].

10) Give an example of a MOVS operation which requires the direction flag to be:

    a) clear b) set

11) How does the CMPS instruction operate? (what does it do, how does it affect the registers and flags, etc.)

12) Which segment contains the source string? The destination string?

13) What is the SCAS instruction used for?

14) How would you quickly initialize an array to all zeros?

15) How are the LODS and STOS instructions used to build complex string operations?

16) How would you use the SUBSTR function to extract a substring of length 6 starting at offset 3 in the StrVar variable, storing the substring in the NewStr variable?

17) What types of errors can occur when the SUBSTR function is executed?

18) Give an example demonstrating the use of each of the following string functions:

    a) INDEX b) REPEAT c) INSERT d) DELETE e) CONCAT

19) Write a short loop which multiplies each element of a single dimensional array by 10. Use the string instructions to fetch and store each array element.

20) The UCR Standard Library does not provide an STRCPYM routine. What is the routine which performs this task?

21) Suppose you are writing an "adventure game" into which the player types sentences and you want to pick out the two words "GO" and "NORTH", if they are present, in the input line. What (non-UCR StdLib) string function appearing in this chapter would you use to search for these words? What UCR Standard Library routine would you use?

22) Explain how to perform an extended precision integer comparison using CMPS

Pattern Matching Chapter 16

The last chapter covered character strings and various operations on those strings. A very typical program reads a sequence of strings from the user and compares the strings to see if they match. For example, DOS' COMMAND.COM program reads command lines from the user and compares the strings the user types to fixed strings like "COPY", "DEL", "RENAME", and so on. Such commands are easy to *parse* because the set of allowable commands is finite and fixed. Sometimes, however, the strings you want to test for are not fixed; instead, they belong to a (possibly infinite) set of different strings. For example, if you execute the DOS command "DEL *.BAK", MS-DOS does not attempt to delete a file named "*.BAK". Instead, it deletes all files which match the *generic pattern* "*.BAK". This, of course, is any file which contains four or more characters and ends with ".BAK". In the MS-DOS world, a string containing characters like "*" and "?" are called *wildcards*; wildcard characters simply provide a way to specify different names via patterns. DOS' wildcard characters are very limited forms of what are known as *regular expressions*; regular expressions are very limited forms of patterns in general. This chapter describes how to create patterns that match a variety of character strings and write pattern matching routines to see if a particular string *matches* a given pattern.

16.1 An Introduction to Formal Language (Automata) Theory

Pattern matching, despite its low-key coverage, is a very important topic in computer science. Indeed, pattern matching is the main programming paradigm in several programming languages like Prolog, SNOBOL4, and Icon. Several programs you use all the time employ pattern matching as a major part of their work. MASM, for example, uses pattern matching to determine if symbols are correctly formed, expressions are proper, and so on. Compilers for high level languages like Pascal and C also make heavy use of pattern matching to parse source files to determine if they are syntactically correct. Surprisingly enough, an important statement known as *Church's Hypothesis* suggests that any computable function can be programmed as a pattern matching problem[1]. Of course, there is no guarantee that the solution would be efficient (they usually are not), but you could arrive at a correct solution. You probably wouldn't need to know about Turing machines (the subject of Church's hypothesis) if you're interested in writing, say, an accounts receivable package. However, there many situations where you may want to introduce the ability to match some generic patterns; so understanding some of the theory of pattern matching is important. This area of computer science goes by the stuffy names of *formal language theory* and *automata theory*. Courses in these subjects are often less than popular because they involve a lot of proofs, mathematics, and, well, theory. However, the concepts behind the proofs are quite simple and very useful. In this chapter we will not bother trying to prove everything about pattern matching. Instead, we will accept the fact that this stuff really works and just apply it. Nonetheless, we do have to discuss some of the results from automata theory, so without further ado…

16.1.1 Machines vs. Languages

You will find references to the term "machine" throughout automata theory literature. This term does not refer to some particular computer on which a program executes. Instead, this is usually some function that reads a string of symbols as input and produces one of two outputs: match or failure. A typical machine (or *automaton*) divides all possible strings into two sets – those strings that it *accepts* (or matches) and those string that it rejects. The *language* accepted by this machine is the set of all strings that the machine

1. Actually, Church's Hypothesis claims that any computable function can be computed on a Turing machine. However, the Turing machine is the ultimate pattern machine computer.

accepts. Note that this language could be infinite, finite, or the empty set (i.e., the machine rejects all input strings). Note that an infinite language does not suggest that the machine accepts all strings. It is quite possible for the machine to accept an infinite number of strings and reject an even greater number of strings. For example, it would be very easy to design a function which accepts all strings whose length is an even multiple of three. This function accepts an infinite number of strings (since there are an infinite number of strings whose length is a multiple of three) yet it rejects twice as many strings as it accepts. This is a very easy function to write. Consider the following 80x86 program that accepts all strings of length three (we'll assume that the carriage return character terminates a string):

```
MatchLen3       proc    near
                getc                    ;Get character #1.
                cmp     al, cr          ;Zero chars if EOLN.
                je      Accept
                getc                    ;Get character #2.
                cmp     al, cr
                je      Failure
                getc                    ;Get character #3.
                cmp     al, cr
                jne     MatchLen3
Failure:        mov     ax, 0           ;Return zero to denote failure.
                ret

Accept:         mov     ax, 1           ;Return one to denote success.
                ret
MatchLen3       endp
```

By tracing through this code, you should be able to easily convince yourself that it returns one in ax if it succeeds (reads a string whose length is a multiple of three) and zero otherwise.

Machines are inherently *recognizers*. The machine itself is the embodiment of a *pattern*. It recognizes any input string which matches the built-in pattern. Therefore, a codification of these automatons is the basic job of the programmer who wants tomatch some patterns.

There are many different classes of machines and the languages they recognize. From simple to complex, the major classifications are *deterministic finite state automata* (which are equivalent to *nondeterministic finite state automata*), *deterministic push down automata, nondeterministic push down automata,* and *Turing machines*. Each successive machine in this list provides a superset of the capabilities of the machines appearing before it. The only reason we don't use Turing machines for everything is because they are more complex to program than, say, a deterministic finite state automaton. If you can match the pattern you want using a deterministic finite state automaton, you'll probably want to code it that way rather than as a Turing machine.

Each class of machine has a class of languages associated with it. Deterministic and nondeterministic finite state automata recognize the *regular* languages. Nondeterministic push down automata recognize the *context free* languages[2]. Turing machines can recognize all recognizable languages. We will discuss each of these sets of languages, and their properties, in turn.

16.1.2 Regular Languages

The regular languages are the least complex of the languages described in the previous section. That does not mean they are less useful; in fact, patterns based on regular expression are probably more common than any other.

2. Deterministic push down automata recognize only a subset of the context free languages.

16.1.2.1 Regular Expressions

The most compact way to specify the strings that belong to a regular language is with a *regular expression*. We shall define, recursively, a regular expression with the following rules:

- ∅ (the empty set) is a regular language and denotes the empty set.
- ε is a regular expression[3]. It denotes the set of languages containing only the empty string: {ε}.
- Any single symbol, *a*, is a regular expression (we will use lower case characters to denote arbitrary symbols). This single symbol matches exactly one character in the input string, that character must be equal to the single symbol in the regular expression. For example, the pattern "m" matches a single "m" character in the input string.

Note that ∅ and ε are not the same. The empty set is a regular language that does not accept *any* strings, including strings of length zero. If a regular language is denoted by {ε}, then it accepts exactly one string, the string of length zero. This latter regular language accepts something, the former does not.

The three rules above provide our *basis* for a recursive definition. Now we will define regular expressions recursively. In the following definitions, assume that *r*, *s*, and *t* are any valid regular expressions.

- Concatenation. If *r* and *s* are regular expressions, so is *rs*. The regular expression *rs* matches any string that begins with a string matched by *r* and ends with a string matched by *s*.
- Alternation/Union. If *r* and *s* are regular expressions, so is *r* | *s* (read this as *r* **or** *s*) This is equivalent to *r* ∪ *s*, (read as *r* union *s*). This regular expression matches any string that *r* or *s* matches.
- Intersection. If *r* and *s* are regular expressions, so is *r* ∩ *s*. This is the set of all strings that both *r* **and** *s* match.
- Kleene Star. If *r* is a regular expression, so is *r**. This regular expression matches zero or more occurrences of *r*. That is, it matches ε, *r*, *rr*, *rrr*, *rrrr*, ...
- Difference. If *r* and *s* are regular expressions, so is *r*-*s*. This denotes the set of strings matched by *r* that are not also matched by *s*.
- Precedence. If *r* is a regular expression, so is (*r*). This matches any string matched by *r* alone. The normal algebraic associative and distributive laws apply here, so (*r* | *s*) *t* is equivalent to *rt* | *st*.

These operators following the normal associative and distributive laws and exhibit the following precedences:

```
Highest:        (r)
                Kleene Star
                Concatentation
                Intersection
                Difference
Lowest:         Alternation/Union
```

Examples:

```
(r | s) t = rt | st
rs* = r(s*)
r ∪ t - s = r ∪ (t - s)
r ∩ t - s = (r ∩ t) - s
```

Generally, we'll use parenthesis to avoid any ambiguity

Although this definition is sufficient for an automata theory class, there are some practical aspects to this definition that leave a little to be desired. For example, to define a

3. The empty string is the string of length zero, containing no symbols.

regular expression that matches a single alphabetic character, you would need to create something like (*a* | *b* | *c* | … | *y* | *z*). Quite a lot of typing for such a trivial character set. Therefore, we shall add some notation to make it easier to specify regular expressions.

- Character Sets. Any set of characters surrounded by brackets, e.g., [abcdefg] is a regular expression and matches a single character from that set. You can specify ranges of characters using a dash, i.e., "[a-z]" denotes the set of lower case characters and this regular expression matches a single lower case character.
- Kleene Plus. If *r* is a regular expression, so is r^+. This regular expression matches one or more occurrences of *r*. That is, it matches *r, rr, rrr, rrrr, …* The precedence of the Kleene Plus is the same as for the Kleene Star. Note that $r^+ = rr^*$.
- Σ represents any single character from the allowable character set. Σ* represents the set of all possible strings. The regular expression Σ*-*r* is the *complement* of *r* – that is, the set of all strings that *r* does not match.

With the notational baggage out of the way, it's time to discuss how to actually use regular expressions as pattern matching specifications. The following examples should give a suitable introduction.

Identifiers: Most programming languages like Pascal or C/C++ specify legal forms for identifiers using a regular expression. Expressed in English terms, the specification is something like "An identifier must begin with an alphabetic character and is followed by zero or more alphanumeric or underscore characters." Using the regular expression (RE) syntax described in this section, an identifier is

[a-zA-Z][a-zA-Z0-9_]*

Integer Consts: A regular expression for integer constants is relatively easy to design. An integer constant consists of an optional plus or minus followed by one or more digits. The RE is

(+ | - | ε) $[0\text{-}9]^+$

Note the use of the empty string (ε) to make the plus or minus optional.

Real Consts: Real constants are a bit more complex, but still easy to specify using REs. Our definition matches that for a real constant appearing in a Pascal program – an optional plus or minus, following by one or more digits; optionally followed by a decimal point and zero or more digits; optionally followed by an "e" or an "E" with an optional sign and one or more digits:

(+ | - | ε) $[0\text{-}9]^+$ ("." [0-9]* | ε) (((e | E) (+ | - | ε) $[0\text{-}9]^+$) | ε)

Since this RE is relatively complex, we should dissect it piece by piece. The first parenthetical term gives us the optional sign. One or more digits are mandatory before the decimal point, the second term provides this. The third term allows an optional decimal point followed by zero or more digits. The last term provides for an optional exponent consisting of "e" or "E" followed by an optional sign and one or more digits.

Reserved Words: It is very easy to provide a regular expression that matches a set of reserved words. For example, if you want to create a regular expression that matches MASM's reserved words, you could use an RE similar to the following:

(mov | add | and | … | mul)

Even: The regular expression (ΣΣ)* matches all strings whose length is a multiple of two.

Sentences: The regular expression:

(Σ* " "*)* run (" "$^+$ (Σ* " "$^+$ | ε)) fast (" " Σ*)*

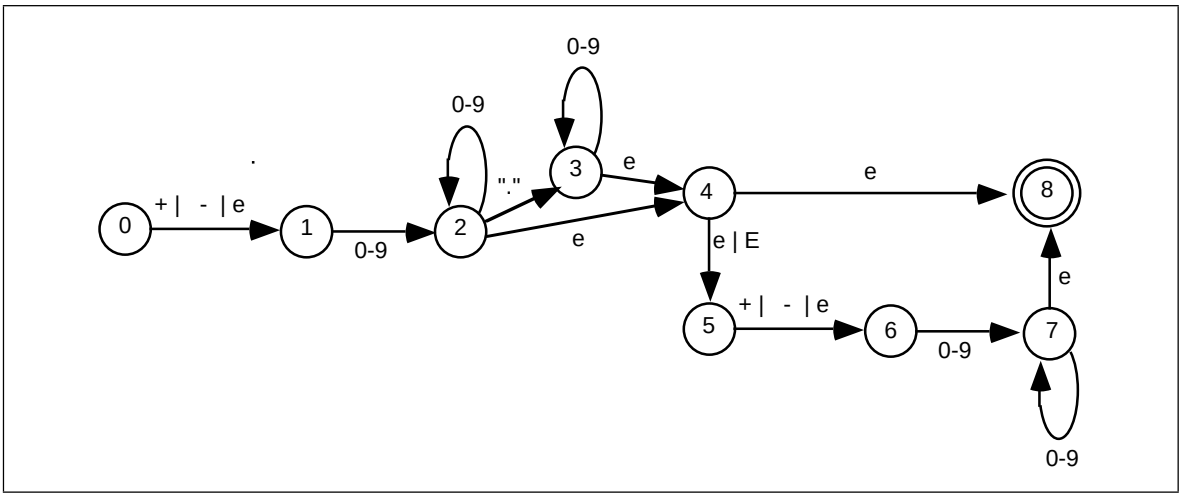


Figure 16.1 NFA for Regular Expression (+ | - | e) [0-9]+ (“.” [0-9]* | e) (((e | E) (+ | - | e) [0-9]+) | e)

matches all strings that contain the separate words “run” followed by “fast” somewhere on the line. This matches strings like “I want to run very fast” and “run as fast as you can” as well as “run fast.”

While REs are convenient for specifying the pattern you want to recognize, they are not particularly useful for creating programs (i.e., “machines”) that actually recognize such patterns. Instead, you should first convert an RE to a *nondeterministic finite state automaton*, or NFA. It is very easy to convert an NFA into an 80x86 assembly language program; however, such programs are rarely efficient as they might be. If efficiency is a big concern, you can convert the NFA into a *deterministic finite state automaton* (DFA) that is also easy to convert to 80x86 assembly code, but the conversion is usually far more efficient.

16.1.2.2 Nondeterministic Finite State Automata (NFAs)

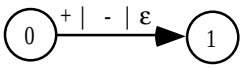
An NFA is a directed graph with *state numbers* associated with each node and *characters or character strings* associated with each edge of the graph. A distinguished state, the *starting state*, determines where the machine begins attempting to match an input string. With the machine in the starting state, it compares input characters against the characters or strings on each edge of the graph. If a set of input characters matches one of the edges, the machine can change states from the node at the start of the edge (the tail) to the state at the end of the edge (the head).

Certain other states, known as *final* or *accepting* states, are usually present as well. If a machine winds up in a final state after exhausting all the input characters, then that machine *accepts* or *matches* that string. If the machine exhausts the input and winds up in a state that is not a final state, then that machine *rejects* the string. Figure 16.1 shows an example NFA for the floating point RE presented earlier.

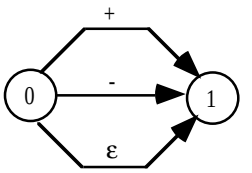
By convention, we’ll always assume that the starting state is state zero. We will denote final states (there may be more than one) by using a double circle for the state (state eight is the final state above).

An NFA always begins with an input string in the starting state (state zero). On each edge coming out of a state there is either ε, a single character, or a character string. To help unclutter the NFA diagrams, we will allow expressions of the form “ xxx | yyy | zzz | …” where xxx, yyy, and zzz are ε, a single character, or a character string. This corresponds to

multiple edges from one state to the other with a single item on each edge. In the example above,



is equivalent to



Likewise, we will allow *sets* of characters, specified by a string of the form x-y, to denote the expression x | x+1 | x+2 | … | y.

Note that an NFA accepts a string if there is *some* path from the starting state to an accepting state that exhausts the input string. There may be multiple paths from the starting state to various final states. Furthermore, there may be some particular path from the starting state to a non-accepting state that exhausts the input string. This does not necessarily mean the NFA rejects that string; if there is some other path from the starting state to an accepting state, then the NFA accepts the string. An NFA rejects a string only if there are *no* paths from the starting state to an accepting state that exhaust the string.

Passing through an accepting state does not cause the NFA to accept a string. You must wind up in a final state *and* exhaust the input string.

To process an input string with an NFA, begin at the starting state. The edges leading out of the starting state will have a character, a string, or ε associated with them. If you choose to move from one state to another along an edge with a single character, then remove that character from the input string and move to the new state along the edge traversed by that character. Likewise, if you choose to move along an edge with a character string, remove that character string from the input string and switch to the new state. If there is an edge with the empty string, ε, then you may elect to move to the new state given by that edge without removing any characters from the input string.

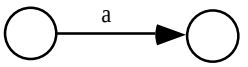
Consider the string “1.25e2” and the NFA in Figure 16.1. From the starting state we can move to state one using the ε string (there is no leading plus or minus, so ε is our only option). From state one we can move to state two by matching the “1” in our input string with the set 0-9; this eats the “1” in our input string leaving “.25e2”. In state two we move to state three and eat the period from the input string, leaving “25e2”. State three loops on itself with numeric input characters, so we eat the “2” and “5” characters at the beginning of our input string and wind up back in state three with a new input string of “e2”. The next input character is “e”, but there is no edge coming out of state three with an “e” on it; there is, however, an ε-edge, so we can use that to move to state four. This move does not change the input string. In state four we can move to state five on an “e” character. This eats the “e” and leaves us with an input string of “2”. Since this is not a plus or minus character, we have to move from state five to state six on the ε edge. Movement from state six to state seven eats the last character in our string. Since the string is empty (and, in particular, it does not contain any digits), state seven cannot loop back on itself. We are currently in state seven (which is not a final state) and our input string is exhausted. However, we can move to state eight (the accepting state) since the transition between states seven and eight is an ε edge. Since we are in a final state and we’ve exhausted the input string, This NFA accepts the input string.

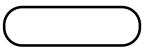
16.1.2.3 Converting Regular Expressions to NFAs

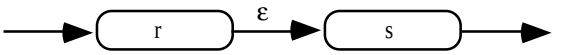
If you have a regular expression and you want to build a machine that recognizes strings in the regular language specified by that expression, you will need to convert the

RE to and NFA. It turns out to be very easy to convert a regular expression to an NFA. To do so, just apply the following rules:

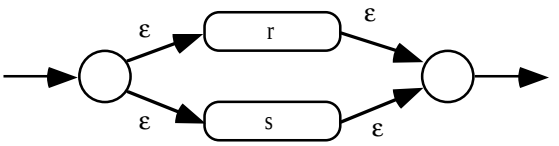
- The NFA representing regular language denoted by the regular expression ∅ (the empty set) is a single, non-accepting state.
- If a regular expression contains an ε, a single character, or a string, create two states and draw an arc between them with ε, the single character, or the string as the label. For example, the RE "a" is converted to an NFA as



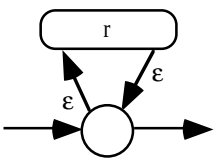
- Let the symbol  denote an NFA which recognizes some regular language specified by some regular expression *r*, *s*, or *t*. If a regular expression takes the form *rs* then the corresponding NFA is



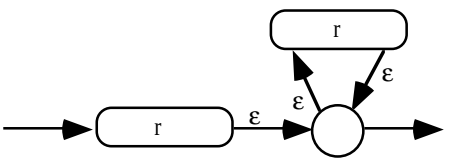
- If a regular expression takes the form *r* | *s*, then the corresponding NFA is



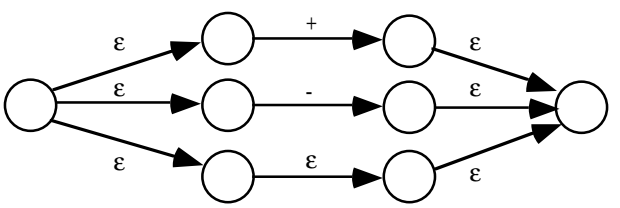
- If a regular expression takes the form *r** then the corresponding NFA is



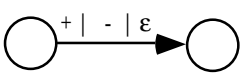
All of the other forms of regular expressions are easily synthesized from these, therefore, converting those other forms of regular expressions to NFAs is a simple two-step process, convert the RE to one of these forms, and then convert this form to the NFA. For example, to convert r^+ to an NFA, you would first convert r^+ to *rr**. This produces the NFA:



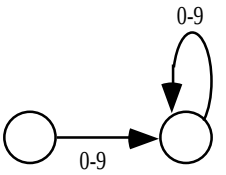
The following example converts the regular expression for an integer constant to an NFA. The first step is to create an NFA for the regular expression (+ | - | ε). The complete construction becomes



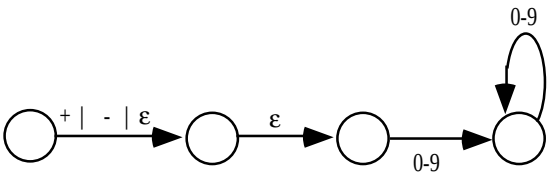
Although we can obviously optimize this to



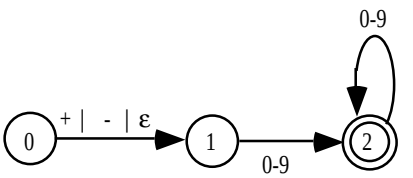
The next step is to handle the $[0\text{-}9]^+$ regular expression; after some minor optimization, this becomes the NFA



Now we simply concatenate the results to produce:



All we need now are starting and final states. The starting state is always the first state of the NFA created by the conversion of the leftmost item in the regular expression. The final state is always the last state of the NFA created by the conversion of the rightmost item in the regular expression. Therefore, the complete regular expression for integer constants (after optimizing out the middle edge above, which serves no purpose) is



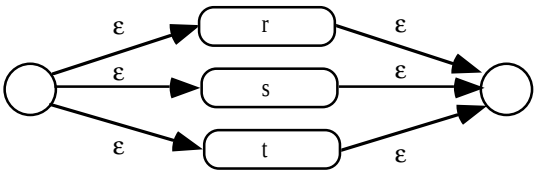
16.1.2.4 Converting an NFA to Assembly Language

There is only one major problem with converting an NFA to an appropriate matching function – NFAs are *nondeterministic*. If you're in some state and you've got some input character, say "a", there is no guarantee that the NFA will tell you what to do next. For example, there is no requirement that edges coming out of a state have unique labels. You could have two or more edges coming out of a state, all leading to different states on the single character "a". If an NFA accepts a string, it only guarantees that there is some path that leads to an accepting state, there is no guarantee that this path will be easy to find.

The primary technique you will use to resolve the nondeterministic behavior of an NFA is *backtracking*. A function that attempts to match a pattern using an NFA begins in the starting state and tries to match the first character(s) of the input string against the edges leaving the starting state. If there is only one match, the code must follow that edge. However, if there are two possible edges to follow, then the code must arbitrarily choose one of them *and remember the others as well as the current point in the input string*. Later, if it turns out the algorithm guessed an incorrect edge to follow, it can return back and try one of the other alternatives (i.e., it *backtracks* and tries a different path). If the algorithm exhausts all alternatives without winding up in a final state (with an empty input string), then the NFA does not accept the string.

Probably the easiest way to implement backtracking is via procedure calls. Let us assume that a matching procedure returns the carry flag set if it succeeds (i.e., accepts a

string) and returns the carry flag clear if it fails (i.e., rejects a string). If an NFA offers multiple choices, you could implement that portion of the NFA as follows:




```
AltRST          proc    near
                push    ax          ;The purpose of these two instructions
                mov     ax, di      ; is to preserve di in case of failure.
                call    r
                jc      Success
                mov     di, ax      ;Restore di (it may be modified by r).
                call    s
                jc      Success
                mov     di, ax      ;Restore di (it may be modified by s).
                call    t
Success:        pop     ax          ;Restore ax.
                ret
AltRST          endp
```

If the r matching procedure succeeds, there is no need to try s and t. On the other hand, if r fails, then we need to try s. Likewise, if r and s both fail, we need to try t. AltRST will fail only if r, s, and t all fail. This code assumes that es:di points at the input string to match. On return, es:di points at the next available character in the string after a match *or it points at some arbitrary point if the match fails.* This code assumes that r, s, and t all preserve the ax register, so it preserves a pointer to the current point in the input string in ax in the event r or s fail.

To handle the individual NFA associated with simple regular expressions (i.e., matching ε or a single character) is not hard at all. Suppose the matching function r matches the regular expression (+ | - | ε). The complete procedure for r is

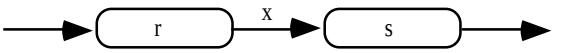
```
r               proc    near
                cmp     byte ptr es:[di], '+'
                je      r_matched
                cmp     byte ptr es:[di], '-'
                jne     r_nomatch
r_matched:      inc     di
r_nomatch:      stc
                ret
r               endp
```

Note that there is no explicit test for ε. If ε is one of the alternatives, the function attempts to match one of the other alternatives first. If none of the other alternatives succeed, then the matching function will succeed anyway, although it does not consume any input characters (which is why the above code skips over the inc di instruction if it does not match "+" or "-"). Therefore, any matching function that has ε as an alternative will always succeed.

Of course, not all matching functions succeed in every case. Suppose the s matching function accepts a single decimal digit. the code for s might be the following:

```
s               proc    near
                cmp     byte ptr es:[di], '0'
                jb      s_fails
                cmp     byte ptr es:[di], '9'
                ja      s_fails
                inc     di
                stc
                ret

s_fails:        clc
                ret
s               endp
```

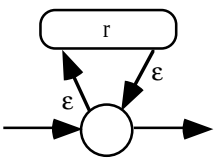
If an NFA takes the form:



Where x is any arbitrary character or string or ε, the corresponding assembly code for this procedure would be

```
ConcatRxS       proc    near
                call    r
                jnc     CRxS_Fail               ;If no r, we won't succeed

; Note, if x=ε then simply delete the following three statements.
; If x is a string rather than a single character, put the the additional
; code to match all the characters in the string.

                cmp     byte ptr es:[di], 'x'
                jne     CRxS_Fail
                inc     di

                call    s
                jnc     CRxS_Fail
                stc                     ;Success!
                ret

CRxS_Fail:      clc
                ret
ConcatRxS       endp
```

If the regular expression is of the form r* and the corresponding NFA is of the form



Then the corresponding 80x86 assembly code can look something like the following:

```
RStar           proc    near
                call    r
                jc      RStar
                stc
                ret
RStar           endp
```

Regular expressions based on the Kleene star always succeed since they allow zero or more occurrences. That is why this code always returns with the carry flag set.

The Kleene Plus operation is only slightly more complex, the corresponding (slightly optimized) assembly code is

```
RPlus           proc    near
                call    r
                jnc     RPlus_Fail
RPlusLp:        call    r
                jc      RPlusLp
                stc
                ret

RPlus_Fail:     clc
                ret
RPlus           endp
```

Note how this routine fails if there isn't at least one occurrence of r.

A major problem with backtracking is that it is potentially inefficient. It is very easy to create a regular expression that, when converted to an NFA and assembly code, generates considerable backtracking on certain input strings. This is further exacerbated by the fact

that matching routines, if written as described above, are generally very short; so short, in fact, that the procedure calls and returns make up a significant portion of the execution time. Therefore, pattern matching in this fashion, although easy, can be slower than it has to be.

This is just a taste of how you would convert REs to NFAs to assembly language. We will not go into further detail in this chapter; not because this stuff isn't interesting to know, but because you will rarely use these techniques in a real program. If you need high performance pattern matching you would not use nondeterministic techniques like these. If you want the ease of programming offered by the conversion of an NFA to assembly language, you still would not use this technique. Instead, the UCR Standard Library provides very powerful pattern matching facilities (which exceed the capabilities of NFAs), so you would use those instead; but more on that a little later.

16.1.2.5 Deterministic Finite State Automata (DFAs)

Nondeterministic finite state automata, when converted to actual program code, may suffer from performance problems because of the backtracking that occurs when matching a string. Deterministic finite state automata solve this problem by comparing different strings *in parallel*. Whereas, in the worst case, an NFA may require *n* comparisons, where *n* is the sum of the lengths of all the strings the NFA recognizes, a DFA requires only *m* comparisons (worst case), where *m* is the length of the longest string the DFA recognizes.

For example, suppose you have an NFA that matches the following regular expression (the set of 80x86 real-mode mnemonics that begin with an "A"):

```
( AAA | AAD | AAM | AAS | ADC | ADD | AND )
```

A typical implementation as an NFA might look like the following:

```
MatchAMnem      proc    near
                strcmpl
                byte    "AAA",0
                je      matched
                strcmpl
                byte    "AAD",0
                je      matched
                strcmpl
                byte    "AAM",0
                je      matched
                strcmpl
                byte    "AAS",0
                je      matched
                strcmpl
                byte    "ADC",0
                je      matched
                strcmpl
                byte    "ADD",0
                je      matched
                strcmpl
                byte    "AND",0
                je      matched
                clc
                ret

matched:        add     di, 3
                stc
                ret
MatchAMnem      endp
```

If you pass this NFA a string that it doesn't match, e.g., "AAND", it must perform seven string comparisons, which works out to about 18 character comparisons (plus all the overhead of calling strcmpl). In fact, a DFA can determine that it does not match this character string by comparing only three characters.

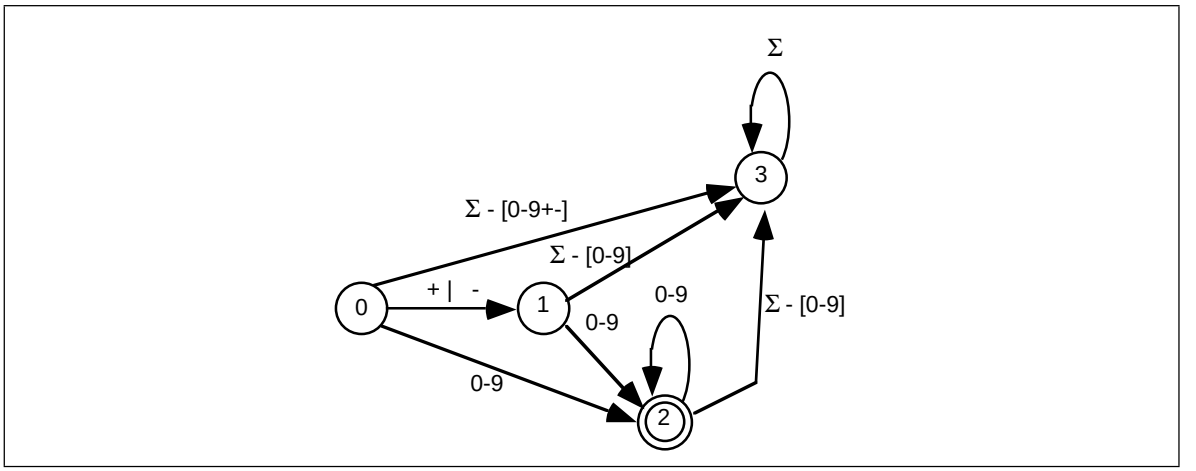


Figure 16.2 DFA for Regular Expression (+ | - | ε) [0-9]$^+$

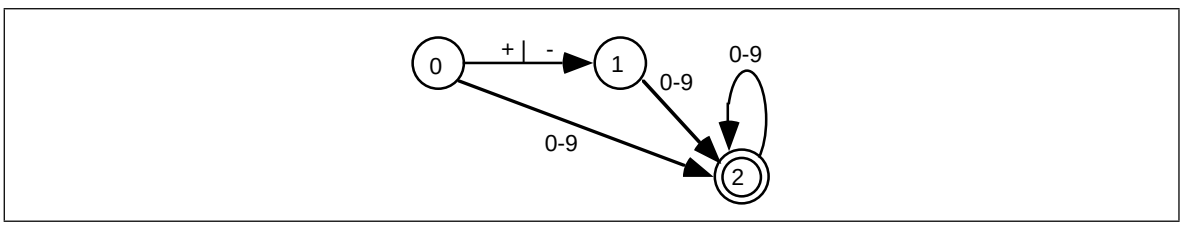


Figure 16.3 Simplified DFA for Regular Expression (+ | - | ε) [0-9]$^+$

A DFA is a special form of an NFA with two restrictions. First, there must be *exactly* one edge coming out of each node for each of the possible input characters; this implies that there must be one edge for each possible input symbol *and* you may not have two edges with the same input symbol. Second, you cannot move from one state to another on the empty string, ε. A DFA is deterministic because at each state the next input symbol determines the next state you will enter. Since each input symbol has an edge associated with it, there is never a case where a DFA "jams" because you cannot leave the state on that input symbol. Similarly, the new state you enter is never ambiguous because there is only one edge leaving any particular state with the current input symbol on it. Figure 16.2 shows the DFA that handles integer constants described by the regular expression

(+ | - | ε) [0-9]$^+$

Note than an expression of the form "Σ - [0-9]" means *any character except a digit*; that is, the *complement* of the set [0-9].

State three is a *failure state*. It is not an accepting state and once the DFA enters a failure state, it is stuck there (i.e., it will consume all additional characters in the input string without leaving the failure state). Once you enter a failure state, the DFA has already rejected the input string. Of course, this is not the only way to reject a string; the DFA above, for example, rejects the empty string (since that leaves you in state zero) and it rejects a string containing only a "+" or a "-" character.

DFAs generally contain more states than a comparable NFA. To help keep the size of a DFA under control, we will allow a few shortcuts that, in no way, affect the operation of a DFA. First, we will remove the restriction that there be an edge associated with each possible input symbol leaving every state. Most of the edges leaving a particular state lead to the failure state. Therefore, our first simplification will be to allow DFAs to drop the edges that lead to a failure state. If a input symbol is not represented on an outgoing edge from some state, we will assume that it leads to a failure state. The above DFA with this simplification appears in Figure 16.2.

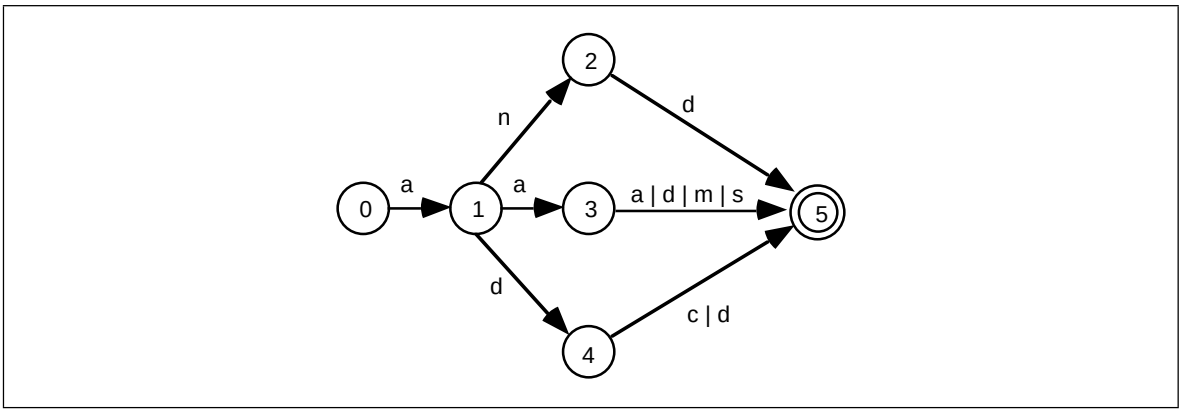
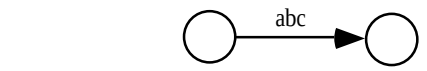
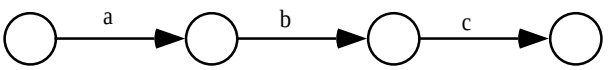


Figure 16.4 DFA that Recognizes AND, AAA, AAD, AAM, AAS, ADD, and ADC

A second shortcut, that is actually present in the two examples above, is to allow sets of characters (or the alternation symbol, "|") to associate several characters with a single edge. Finally, we will also allow strings attached to an edge. This is a shorthand notation for a list of states which recognize each successive character, i.e., the following two DFAs are equivalent:





Returning to the regular expression that recognizes 80x86 real-mode mnemonics beginning with an "A", we can construct a DFA that recognizes such strings as shown in Figure 16.4.

If you trace through this DFA by hand on several accepting and rejecting strings, you will discover than it requires no more than six character comparisons to determine whether the DFA should accept or reject an input string.

Although we are not going to discuss the specifics here, it turns out that regular expressions, NFAs, and DFAs are all equivalent. That is, you can convert anyone of these to the others. In particular, you can always convert an NFA to a DFA. Although the conversion isn't totally trivial, especially if you want an *optimized* DFA, it is always possible to do so. Converting between all these forms is beginning to leave the scope of this text. If you are interested in the details, *any* text on formal languages or automata theory will fill you in.

16.1.2.6 Converting a DFA to Assembly Language

It is relatively straightforward to convert a DFA to a sequence of assembly instructions. For example, the assembly code for the DFA that accepts the A-mnemonics in the previous section is

```
DFA_A_Mnem      proc    near
                cmp     byte ptr es:[di], 'A'
                jne     Fail
                cmp     byte ptr es:[di+1], 'A'
                je      DoAA
                cmp     byte ptr es:[di+1], 'D'
                je      DoAD
                cmp     byte ptr es:[di+1], 'N'
                je      DoAN
```

```
Fail:           clc
                ret

DoAN:           cmp     byte ptr es:[di+2], 'D'
                jne     Fail
Succeed:        add     di, 3
                stc
                ret

DoAD:           cmp     byte ptr es:[di+2], 'D'
                je      Succeed
                cmp     byte ptr es:[di+2], 'C'
                je      Succeed
                clc                                     ;Return Failure
                ret

DoAA:           cmp     byte ptr es:[di+2], 'A'
                je      Succeed
                cmp     byte ptr es:[di+2], 'D'
                je      Succeed
                cmp     byte ptr es:[di+2], 'M'
                je      Succeed
                cmp     byte ptr es:[di+2], 'S'
                je      Succeed
                clc
                ret
DFA_A_Mnem      endp
```

Although this scheme works and is considerably more efficient than the coding scheme for NFAs, writing this code can be tedious, especially when converting a large DFA to assembly code. There is a technique that makes converting DFAs to assembly code almost trivial, although it can consume quite a bit of space – to use state machines. A simple state machine is a two dimensional array. The columns are indexed by the possible characters in the input string and the rows are indexed by state number (i.e., the states in the DFA). Each element of the array is a new state number. The algorithm to match a given string using a state machine is trivial, it is

```
state := 0;
while (another input character ) do begin

       ch := next input character ;
       state := StateTable [state][ch];

end;
if (state in FinalStates) then accept
else reject;
```

FinalStates is a set of accepting states. If the current state number is in this set after the algorithm exhausts the characters in the string, then the state machine accepts the string, otherwise it rejects the string.

The following state table corresponds to the DFA for the "A" mnemonics appearing in the previous section:

Table 62: State Machine for 80x86 “A” Instructions DFA

State	A	C	D	M	N	S	Else
0	1	F	F	F	F	F	F
1	3	F	4	F	2	F	F
2	F	F	5	F	F	F	F
3	5	F	5	5	F	5	F
4	F	5	5	F	F	F	F
5	F	F	F	F	F	F	F
F	F	F	F	F	F	F	F

State five is the only accepting state.

There is one major drawback to using this table driven scheme – the table will be quite large. This is not apparent in the table above because the column labelled “Else” hides considerable detail. In a true state table, you will need one column for each possible input character. since there are 256 possible input characters (or at least 128 if you’re willing to stick to seven bit ASCII), the table above will have 256 columns. With only one byte per element, this works out to about 2K for this small state machine. Larger state machines could generate very large tables.

One way to reduce the size of the table at a (very) slight loss in execution speed is to classify the characters before using them as an index into a state table. By using a single 256-byte lookup table, it is easy to reduce the state machine to the table above. Consider the 256 byte lookup table that contains:

- A one at positions *Base+”a”* and *Base+”A”*,
- A two at locations *Base+”c”* and *Base+”C”*,
- A three at locations *Base+”d”* and *Base+”D”*,
- A four at locations *Base+”m”* and *Base+”M”*,
- A five at locations *Base+”n”* and *Base+”N”*,
- A six at locations *Base+”s”* and *Base+”S”*, and
- A zero everywhere else.

Now we can modify the above table to produce:

Table 63: Classified State Machine Table for 80x86 “A” Instructions DFA

State	0	1	2	3	4	5	6	7
0	6	1	6	6	6	6	6	6
1	6	3	6	4	6	2	6	6
2	6	6	6	5	6	6	6	6
3	6	5	6	5	5	6	5	6
4	6	6	5	5	6	6	6	6
5	6	6	6	6	6	6	6	6
6	6	6	6	6	6	6	6	6

The table above contains an extra column, “7”, that we will not use. The reason for adding the extra column is to make it easy to index into this two dimensional array (since the extra column lets us multiply the state number by eight rather than seven).

Assuming Classify is the name of the lookup table, the following 80386 code recognizes the strings specified by this DFA:

```
DFA2_A_Mnem     proc
                push    ebx                             ;Ptr to Classify.
                push    eax                             ;Current character.
                push    ecx                             ;Current state.
                xor     eax, eax                        ;EAX := 0
                mov     ebx, eax                        ;EBX := 0
                mov     ecx, eax                        ;ECX (state) := 0
                lea     bx, Classify
WhileNotEOS:    mov     al, es:[di]                     ;Get next input char.
                cmp     al, 0                           ;At end of string?
                je      AtEOS
                xlat                                    ;Classify character.
                mov     cl, State_Tbl[eax+ecx*8]        ;Get new state #.
                inc     di                              ;Move on to next char.
                jmp     WhileNotEOS

AtEOS:          cmp     cl, 5                           ;In accepting state?
                stc                                     ;Assume acceptance.
                je      Accept
                clc
Accept:         pop     ecx
                pop     eax
                pop     ebx
                ret
DFA2_A_Mnem     endp
```

The nice thing about this DFA (the DFA is the combination of the classification table, the state table, and the above code) is that it is very easy to modify. To handle any other state machine (with eight or fewer character classifications) you need only modify the Classification array, the State_Tbl array, the lea bx, Classify statement and the statements at label AtEOS that determine if the machine is in a final state. The assembly code does not get more complex as the DFA grows in size. The State_Tbl array will get larger as you add more states, but this does not affect the assembly code.

Of course, the assembly code above *does* assume there are exactly eight columns in the matrix. It is easy to generalize this code by inserting an appropriate imul instruction to multiply by the size of the array. For example, had we gone with seven columns rather than eight, the code above would be

```
DFA2_A_Mnem     proc
                push    ebx                             ;Ptr to Classify.
                push    eax                             ;Current character.
                push    ecx                             ;Current state.
                xor     eax, eax                        ;EAX := 0
                mov     ebx, eax                        ;EBX := 0
                mov     ecx, eax                        ;ECX (state) := 0
                lea     bx, Classify
WhileNotEOS:    mov     al, es:[di]                     ;Get next input char.
                cmp     al, 0                           ;At end of string?
                je      AtEOS
                xlat                                    ;Classify character.
                imul    cx, 7
                movzx   ecx, State_Tbl[eax+ecx]         ;Get new state #.
                inc     di                              ;Move on to next char.
                jmp     WhileNotEOS

AtEOS:          cmp     cl, 5                           ;In accepting state?
                stc                                     ;Assume acceptance.
                je      Accept
                clc
Accept:         pop     ecx
                pop     eax
                pop     ebx
                ret
DFA2_A_Mnem     endp
```

Although using a state table in this manner simplifies the assembly coding, it does suffer from two drawbacks. First, as mentioned earlier, it is slower. This technique has to

execute all the statements in the while loop for each character it matches; and those instructions are not particularly fast ones, either. The second drawback is that you've got to create the state table for the state machine; that process is tedious and error prone.

If you need the absolute highest performance, you can use the state machine techniques described in (see "State Machines and Indirect Jumps" on page 529). The trick here is to represent each state with a short segment of code and its own one dimensional state table. Each entry in the table is the target address of the segment of code representing the next state. The following is an example of our "A Mnemonic" state machine written in this fashion. The only difference is that the zero byte is classified to value seven (zero marks the end of the string, we will use this to determine when we encounter the end of the string). The corresponding state table would be:

Table 64: Another State Machine Table for 80x86 "A" Instructions DFA

State	0	1	2	3	4	5	6	7
0	6	1	6	6	6	6	6	6
1	6	3	6	4	6	2	6	6
2	6	6	6	5	6	6	6	6
3	6	5	6	5	5	6	5	6
4	6	6	5	5	6	6	6	6
5	6	6	6	6	6	6	6	5
6	6	6	6	6	6	6	6	6

The 80x86 code is

```
DFA3_A_Mnem     proc
                push    ebx
                push    eax
                push    ecx
                xor     eax, eax

                lea     ebx, Classify
State0:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State0Tbl[eax*2]

State0Tbl       word    State6, State1, State6, State6
                word    State6, State6, State6, State6

State1:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State1Tbl[eax*2]

State1Tbl       word    State6, State3, State6, State4
                word    State6, State2, State6, State6

State2:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State2Tbl[eax*2]

State2Tbl       word    State6, State6, State6, State5
                word    State6, State6, State6, State6

State3:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State3Tbl[eax*2]
```

```
State3Tbl       word    State6, State5, State6, State5
                word    State5, State6, State5, State6

State4:         mov     al, es:[di]
                xlat
                inc     di
                jmp     cseg:State4Tbl[eax*2]

State4Tbl       word    State6, State6, State5, State5
                word    State6, State6, State6, State6

State5:         mov     al, es:[di]
                cmp     al, 0
                jne     State6
                stc
                pop     ecx
                pop     eax
                pop     ebx
                ret

State6:         clc
                pop     ecx
                pop     eax
                pop     ebx
                ret
```

There are two important features you should note about this code. First, it only executes four instructions per character comparison (fewer, on the average, than the other techniques). Second, the instant the DFA detects failure it stops processing the input characters. The other table driven DFA techniques blindly process the entire string, even after it is obvious that the machine is locked in a failure state.

Also note that this code treats the accepting and failure states a little differently than the generic state table code. This code recognizes the fact that once we're in state five it will either succeed (if EOS is the next character) or fail. Likewise, in state six this code knows better than to try searching any farther.

Of course, this technique is not as easy to modify for different DFAs as a simple state table version, but it is quite a bit faster. If you're looking for speed, this is a good way to code a DFA.

16.1.3 Context Free Languages

Context free languages provide a superset of the regular languages – if you can specify a class of patterns with a regular expression, you can express the same language using a *context free grammar*. In addition, you can specify many languages that are not regular using context free grammars (CFGs).

Examples of languages that are context free, but not regular, include the set of all strings representing common arithmetic expressions, legal Pascal or C source files[4], and MASM macros. Context free languages are characterized by *balance* and *nesting*. For example, arithmetic expression have balanced sets of parenthesis. High level language statements like repeat…until allow nesting and are always balanced (e.g., for every repeat there is a corresponding until statement later in the source file).

There is only a slight extension to the regular languages to handle context free languages – function calls. In a regular expression, we only allow the objects we want to match and the specific RE operators like "|", "*", concatenation, and so on. To extend regular languages to context free languages, we need only add recursive function calls to regular expressions. Although it would be simple to create a syntax allowing function calls

4. Actually, C and Pascal are *not* context free languages, but Computer Scientists like to treat them as though they were.

within a regular expression, computer scientists use a different notation altogether for context free languages – a context free grammar.

A context free grammar contains two types of symbols: *terminal symbols* and *nonterminal symbols*. Terminal symbols are the individual characters and strings that the context free grammar matches plus the empty string, ε. Context free grammars use nonterminal symbols for function calls and definitions. In our context free grammars we will use italic characters to denote nonterminal symbols and standard characters to denote terminal symbols.

A context free grammar consists of a set of function definitions known as *productions*. A production takes the following form:

Function_Name → «list of terminal and nonterminal symbols»

The function name to the left hand side of the arrow is called the *left hand side* of the production. The function body, which is the list of terminals and nonterminal symbols, is called the *right hand side* of the production. The following is a grammar for simple arithmetic expressions:

expression → *expression* + *factor*
expression → *expression* - *factor*
expression → *factor*
factor → *factor* * *term*
factor → *factor* / *term*
factor → *term*
term → *IntegerConstant*
term → (*expression*)
IntegerConstant → *digit*
IntegerConstant → *digit IntegerConstant*
digit → 0
digit → 1
digit → 2
digit → 3
digit → 4
digit → 5
digit → 6
digit → 7
digit → 8
digit → 9

Note that you may have multiple definitions for the same function. Context-free grammars behave in a non-deterministic fashion, just like NFAs. When attempting to match a string using a context free grammar, a string matches if there exists some matching function which matches the current input string. Since it is very common to have multiple productions with identical left hand sides, we will use the alternation symbol from the regular expressions to reduce the number of lines in the grammar. The following two subgrammars are identical:

expression → *expression* + *factor*
expression → *expression* - *factor*
expression → *factor*

The above is equivalent to:

expression → *expression* + *factor* | *expression* - *factor* | *factor*

The full arithmetic grammar, using this shorthand notation, is

expression → *expression* + *factor* | *expression* - *factor* | *factor*
factor → *factor* * *term* | *factor* / *term* | *term*
term → *IntegerConstant* | (*expression*)

IntegerConstant → *digit* | *digit IntegerConstant*
digit → 0 | 1 | 2 | 3 | *4* | 5 | 6 | 7 | 8 | 9

One of the nonterminal symbols, usually the first production in the grammar, is the *starting symbol*. This is roughly equivalent to the starting state in a finite state automaton. The starting symbol is the first matching function you call when you want to test some input string to see if it is a member of a context free language. In the example above, *expression* is the starting symbol.

Much like the NFAs and DFAs recognize strings in a regular language specified by a regular expression, *nondeterministic pushdown automata* and *deterministic pushdown automata* recognize strings belonging to a context free language specified by a context free grammar. We will not go into the details of these pushdown automata (or *PDAs*) here, just be aware of their existence. We can match strings directly with a grammar. For example, consider the string

7+5*(2+1)

To match this string, we begin by calling the starting symbol function, *expression*, using the function *expression* → *expression* + *factor*. The first plus sign suggests that the *expression* term must match "7" and the *factor* term must match "5*(2+1)". Now we need to match our input string with the pattern *expression* + *factor*. To do this, we call the *expression* function once again, this time using the *expression* → *factor* production. This give us the *reduction*:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor*

The ⇒ symbol denotes the application of a nonterminal function call (a reduction).

Next, we call the factor function, using the production *factor* → *term* to yield the reduction:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor*

Continuing, we call the *term* function to produce the reduction:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor* ⇒ *IntegerConstant* + *factor*

Next, we call the *IntegerConstant* function to yield:

expression ⇒ *expression* + *factor* ⇒ *factor* + *factor* ⇒ *term* + *factor* ⇒ *IntegerConstant* + *factor* ⇒ *7* + *factor*

At this point, the first two symbols of our generated string match the first two characters of the input string, so we can remove them from the input and concentrate on the items that follow. In succession, we call the *factor* function to produce the reduction *7* + *factor* * *term* and then we call *factor, term*, and *IntegerConstant* to yield *7* + *5* * *term*. In a similar fashion, we can reduce the term to "(*expression*)" and reduce expression to "2+1". The complete *derivation* for this string is

$$
\begin{aligned}
\textit{expression} &\Rightarrow \textit{expression} + \textit{factor} \\
&\Rightarrow \textit{factor} + \textit{factor} \\
&\Rightarrow \textit{term} + \textit{factor} \\
&\Rightarrow \textit{IntegerConstant} + \textit{factor} \\
&\Rightarrow \textit{7} + \textit{factor} \\
&\Rightarrow \textit{7} + \textit{factor} * \textit{term} \\
&\Rightarrow \textit{7} + \textit{term} * \textit{term} \\
&\Rightarrow \textit{7} + \textit{IntegerConstant} * \textit{term} \\
&\Rightarrow \textit{7} + \textit{5} * \textit{term} \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{expression}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{expression} + \textit{factor}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{factor} + \textit{factor}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{IntegerConstant} + \textit{factor}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{2} + \textit{factor}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{2} + \textit{term}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{2} + \textit{IntegerConstant}) \\
&\Rightarrow \textit{7} + \textit{5} * (\textit{2} + \textit{1})
\end{aligned}
$$

The final reduction completes the derivation of our input string, so the string 7+5*(2+1) is in the language specified by the context free grammar.

16.1.4 Eliminating Left Recursion and Left Factoring CFGs

In the next section we will discuss how to convert a CFG to an assembly language program. However, the technique we are going to use to do this conversion will require that we modify certain grammars before converting them. The arithmetic expression grammar in the previous section is a good example of such a grammar – one that is *left recursive*.

Left recursive grammars pose a problem for us because the way we will typically convert a production to assembly code is to call a function corresponding to a nonterminal and compare against the terminal symbols. However, we will run into trouble if we attempt to convert a production like the following using this technique:

$$\textit{expression} \rightarrow \textit{expression} + \textit{factor}$$

Such a conversion would yield some assembly code that looks roughly like the following:

```
expression      proc    near
                call    expression
                jnc     fail
                cmp     byte ptr es:[di], '+'
                jne     fail
                inc     di
                call    factor
                jnc     fail
                stc
                ret
Fail:           clc
                ret
expression      endp
```

The obvious problem with this code is that it will generate an infinite loop. Upon entering the expression function this code immediately calls expression recursively, which immediately calls expression recursively, which immediately calls expression recursively, ... Clearly, we need to resolve this problem if we are going to write any real code to match this production.

The trick to resolving left recursion is to note that if there is a production that suffers from left recursion, there must be *some* production with the same left hand side that is not left recursive[5]. All we need do is rewrite the left recursive call in terms of the production

that does not have any left recursion. This sound like a difficult task, but it's actually quite easy.

To see how to eliminate left recursion, let X_i and Y_j represent any set of terminal symbols or nonterminal symbols that do not have a right hand side beginning with the nonterminal *A*. If you have some productions of the form:

$$A \rightarrow AX_1 \mid AX_2 \mid \ldots \mid AX_n \mid Y_1 \mid Y_2 \mid \ldots \mid Y_m$$

You will be able to translate this to an equivalent grammar without left recursion by replacing each term of the form $A \rightarrow Y_i$ by $A \rightarrow Y_i A$ and each term of the form $A \rightarrow AX_i$ by $A' \rightarrow X_i A' \mid \varepsilon$. For example, consider three of the productions from the arithmetic grammar:

expression → *expression* + *factor*
expression → *expression* - *factor*
expression → *factor*

In this example *A* corresponds to *expression*, X_1 corresponds to "+ *factor* ", X_2 corresponds to "- *factor* ", and Y_1 corresponds to "*factor* ". The equivalent grammar without left recursion is

expression → *factor E'*
E' → - *factor E'*
E' → + *factor E'*
E' → ε

The complete arithmetic grammar, with left recursion removed, is

expression → *factor E'*
E' → + *factor E'* | - *factor E'* | ε
factor → *term F'*
F' → * *term F'* | / *term F'* | ε
term → *IntegerConstant* | (*expression*)
IntegerConstant → *digit* | *digit IntegerConstant*
digit → 0 | 1 | 2 | 3 | *4* | 5 | 6 | 7 | 8 | 9

Another useful transformation on a grammar is to left factor the grammar. This can reduce the need for backtracking, improving the performance of your pattern matching code. Consider the following CFG fragment:

stmt → if *expression* then *stmt* endif
stmt → if *expression* then *stmt* else *stmt* endif

These two productions begin with the same set of symbols. Either production will match all the characters in an if statement up to the point the matching algorithm encounters the first else or endif. If the matching algorithm processes the first statement up to the point of the endif terminal symbol and encounters the else terminal symbol instead, it must backtrack all the way to the if symbol and start over. This can be terribly inefficient because of the recursive call to *stmt* (imagine a 10,000 line program that has a single if statement around the entire 10,000 lines, a compiler using this pattern matching technique would have to recompile the entire program from scratch if it used backtracking in this fashion). However, by left factoring the grammar before converting it to program code, you can eliminate the need for backtracking.

To left factor a grammar, you collect all productions that have the same left hand side and begin with the same symbols on the right hand side. In the two productions above, the common symbols are "if *expression* then *stmt* ". You combine the common strings into a single production and then append a new nonterminal symbol to the end of this new production, e.g.,

5. If this is not the case, the grammar does not match any finite length strings.

stmt → if *expression* then *stmt NewNonTerm*

Finally, you create a new set of productions using this new nonterminal for each of the suffixes to the common production:

NewNonTerm → endif | else *stmt* endif

This eliminates backtracking because the matching algorithm can process the if, the *expression*, the then, and the stmt before it has to choose between endif and else.

16.1.5 Converting REs to CFGs

Since the context free languages are a superset of the regular languages, it should come as no surprise that it is possible to convert regular expressions to context free grammars. Indeed, this is a very easy process involving only a few intuitive rules.

1) If a regular expression simply consists of a sequence of characters, xyz, you can easily create a production for this regular expression of the form *P* → xyz. This applies equally to the empty string, ε.

2) If *r* and *s* are two regular expression that you've converted to CFG productions *R* and *S*, and you have a regular expression *rs* that you want to convert to a production, simply create a new production of the form *T* → *R S*.

3) If *r* and *s* are two regular expression that you've converted to CFG productions *R* and *S*, and you have a regular expression *r* | *s* that you want to convert to a production, simply create a new production of the form *T* → *R* | *S*.

4) If *r* is a regular expression that you've converted to a production, *R*, and you want to create a production for *r**, simply use the production *RStar* → *R RStar* | ε.

5) If *r* is a regular expression that you've converted to a production, *R*, and you want to create a production for r^+, simply use the production *RPlus* → *R RPlus* | *R*.

6) For regular expressions there are operations with various precedences. Regular expressions also allow parenthesis to override the default precedence. This notion of precedence does not carry over into CFGs. Instead, you must encode the precedence directly into the grammar. For example, to encode *R S** you would probably use productions of the form:

```
T → R SStar
SStar → S SStar | ε
```

Likewise, to handle a grammar of the form (*RS*)* you could use productions of the form:

```
T → R S T | ε
RS → R S
```

16.1.6 Converting CFGs to Assembly Language

If you have removed left recursion and you've left factored a grammar, it is very easy to convert such a grammar to an assembly language program that recognizes strings in the context free language.

The first convention we will adopt is that es:di always points at the start of the string we want to match. The second convention we will adopt is to create a function for each nonterminal. This function returns success (carry set) if it matches an associated subpattern, it returns failure (carry clear) otherwise. If it succeeds, it leaves di pointing at the next character is the staring *after* the matched pattern; if it fails, it preserves the value in di across the function call.

To convert a set of productions to their corresponding assembly code, we need to be able to handle four things: terminal symbols, nonterminal symbols, alternation, and the

empty string. First, we will consider simple functions (nonterminals) which do not have multiple productions (i.e., alternation).

If a production takes the form $T \rightarrow \varepsilon$ and there are no other productions associated with *T*, then this production always succeeds. The corresponding assembly code is simply:

```
T               proc    near
                stc
                ret
T               endp
```

Of course, there is no real need to ever call *T* and test the returned result since we know it will always succeed. On the other hand, if *T* is a *stub* that you intend to fill in later, you should call *T*.

If a production takes the form $T \rightarrow$ xyz, where xyz is a string of one or more terminal symbols, then the function returns success if the next several input characters match xyz, it returns failure otherwise. Remember, if the prefix of the input string matches xyz, then the matching function must advance di beyond these characters. If the first characters of the input string does not match xyz, it must preserve di. The following routines demonstrate two cases, where xyz is a single character and where xyz is a string of characters:

```
T1              proc    near
                cmp     byte ptr es:[di], 'x'       ;Single char.
                je      Success
                clc                                 ;Return Failure.
                ret

Success:        inc     di                          ;Skip matched char.
                stc                                 ;Return success.
                ret
T1              endp


T2              proc    near
                call    MatchPrefix
                byte    'xyz',0
                ret
T2              endp
```

MatchPrefix is a routine that matches the prefix of the string pointed at by es:di against the string following the call in the code stream. It returns the carry set and adjusts di if the string in the code stream is a prefix of the input string, it returns the carry flag clear and preserves di if the literal string is not a prefix of the input. The MatchPrefix code follows:

```
MatchPrefix     proc    far         ;Must be far!
                push    bp
                mov     bp, sp
                push    ax
                push    ds
                push    si
                push    di

                lds     si, 2[bp]   ;Get the return address.
CmpLoop:        mov     al, ds:[si] ;Get string to match.
                cmp     al, 0       ;If at end of prefix,
                je      Success     ; we succeed.
                cmp     al, es:[di] ;See if it matches prefix,
                jne     Failure     ; if not, immediately fail.
                inc     si
                inc     di
                jmp     CmpLoop

Success:        add     sp, 2       ;Don't restore di.
                inc     si          ;Skip zero terminating byte.
                mov     2[bp], si   ;Save as return address.
                pop     si
                pop     ds
                pop     ax
```

```
                pop     bp
                stc                     ;Return success.
                ret

Failure:        inc     si                      ;Need to skip to zero byte.
                cmp     byte ptr ds:[si], 0
                jne     Failure
                inc     si
                mov     2[bp], si               ;Save as return address.

                pop     di
                pop     si
                pop     ds
                pop     ax
                pop     bp
                clc                             ;Return failure.
                ret
MatchPrefix     endp
```

If a production takes the form $T \rightarrow R$, where R is a nonterminal, then the T function calls R and returns whatever status R returns, e.g.,

```
T               proc    near
                call    R
                ret
T               endp
```

If the right hand side of a production contains a string of terminal and nonterminal symbols, the corresponding assembly code checks each item in turn. If any check fails, then the function returns failure. If all items succeed, then the function returns success. For example, if you have a production of the form $T \rightarrow R$ abc S you could implement this in assembly language as

```
T               proc    near
                push    di                      ;If we fail, must preserve
di.
                call    R
                jnc     Failure
                call    MatchPrefix
                byte    "abc",0
                jnc     Failure
                call    S
                jnc     Failure
                add     sp, 2                   ;Don't preserve di if we
succeed.
                stc
                ret

Failure:        pop     di
                clc
                ret
T               endp
```

Note how this code preserves di if it fails, but does not preserve di if it succeeds.

If you have multiple productions with the same left hand side (i.e., alternation), then writing an appropriate matching function for the productions is only slightly more complex than the single production case. If you have multiple productions associated with a single nonterminal on the left hand side, then create a sequence of code to match each of the individual productions. To combine them into a single matching function, simply write the function so that it succeeds if any one of these code sequences succeeds. If one of the productions is of the form T $\rightarrow$ e, then test the other conditions first. If none of them could be selected, the function succeeds. For example, consider the productions:

$$E' \rightarrow + \ factor \ E' \ | \ - \ factor \ E' \ | \ \varepsilon$$

This translates to the following assembly code:

```
EPrime          proc        near
                push        di
                cmp         byte ptr es:[di], '+'
                jne         TryMinus
                inc         di
                call        factor
                jnc         EP_Failed
                call        EPrime
                jnc         EP_Failed
Success:        add         sp, 2
                stc
                ret

TryMinus:       cmp         byte ptr es:[di], '-'
                jne         EP_Failed
                inc         di
                call        factor
                jnc         EP_Failed
                call        EPrime
                jnc         EP_Failed
                add         sp, 2
                stc
                ret

EP_Failed:      pop         di
                stc                         ;Succeed because of E' -> ε
                ret
EPrime          endp
```

This routine always succeeds because it has the production $E' \rightarrow \varepsilon$. This is why the stc instruction appears after the EP_Failed label.

To invoke a pattern matching function, simply load es:di with the address of the string you want to test and call the pattern matching function. On return, the carry flag will contain one if the pattern matches the string up to the point returned in di. If you want to see if the entire string matches the pattern, simply check to see if es:di is pointing at a zero byte when you get back from the function call. If you want to see if a string belongs to a context free language, you should call the function associated with the starting symbol for the given context free grammar.

The following program implements the arithmetic grammar we've been using as examples throughout the past several sections. The complete implementation is

```
; ARITH.ASM
;
; A simple recursive descent parser for arithmetic strings.

                .xlist
                include  stdlib.a
                includelibstdlib.lib
                .list


dseg            segment  para public 'data'

; Grammar for simple arithmetic grammar (supports +, -, *, /):
;
; E -> FE'
; E' -> + F E' | - F E' | <empty string>
; F -> TF'
; F' -> * T F' | / T F' | <empty string>
; T -> G | (E)
; G -> H | H G
; H -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;


InputLine       byte     128 dup (0)

dseg            ends
```

```
cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Matching functions for the grammar.
; These functions return the carry flag set if they match their
; respective item. They return the carry flag clear if they fail.
; If they fail, they preserve di. If they succeed, di points to
; the first character after the match.


; E -> FE'

E               proc     near
                push     di
                call     F           ;See if F, then E', succeeds.
                jnc      E_Failed
                call     EPrime
                jnc      E_Failed
                add      sp, 2       ;Success, don't restore di.
                stc
                ret

E_Failed:       pop      di          ;Failure, must restore di.
                clc
                ret
E               endp



; E' -> + F E' | - F E' | ε

EPrime          proc     near
                push     di

; Try + F E' here

                cmp      byte ptr es:[di], '+'
                jne      TryMinus
                inc      di
                call     F
                jnc      EP_Failed
                call     EPrime
                jnc      EP_Failed
Success:        add      sp, 2
                stc
                ret

; Try  - F E' here.

TryMinus:       cmp      byte ptr es:[di], '-'
                jne      Success
                inc      di
                call     F
                jnc      EP_Failed
                call     EPrime
                jnc      EP_Failed
                add      sp, 2
                stc
                ret

; If none of the above succeed, return success anyway because we have
; a production of the form E' -> ε.

EP_Failed:      pop      di
                stc
                ret
EPrime          endp
```

```
; F -> TF'

F               proc    near
                push    di
                call    T
                jnc     F_Failed
                call    FPrime
                jnc     F_Failed
                add     sp, 2       ;Success, don't restore di.
                stc
                ret

F_Failed:       pop     di
                clc
                ret
F               endp



; F -> * T F' | / T F' | ε

FPrime          proc    near
                push    di
                cmp     byte ptr es:[di], '*'       ;Start with "*"?
                jne     TryDiv
                inc     di                          ;Skip the "*".
                call    T
                jnc     FP_Failed
                call    FPrime
                jnc     FP_Failed
Success:        add     sp, 2
                stc
                ret

; Try F -> / T F' here

TryDiv:         cmp     byte ptr es:[di], '/'       ;Start with "/"?
                jne     Success                     ;Succeed anyway.
                inc     di                          ;Skip the "/".
                call    T
                jnc     FP_Failed
                call    FPrime
                jnc     FP_Failed
                add     sp, 2
                stc
                ret

; If the above both fail, return success anyway because we've got
; a production of the form F -> ε

FP_Failed:      pop     di
                stc
                ret
FPrime          endp


; T -> G | (E)

T               proc    near

; Try T -> G here.

                call    G
                jnc     TryParens
                ret

; Try T -> (E) here.
```

```
TryParens:      push    di                          ;Preserve if we fail.
                cmp     byte ptr es:[di], '('       ;Start with "("?
                jne     T_Failed                    ;Fail if no.
                inc     di                          ;Skip "(" char.
                call    E
                jnc     T_Failed
                cmp     byte ptr es:[di], ')'       ;End with ")"?
                jne     T_Failed                    ;Fail if no.
                inc     di                          ;Skip ")"
                add     sp, 2                       ;Don't restore di,
                stc                                 ; we've succeeded.
                ret

T_Failed:       pop     di
                clc
                ret
T               endp


; The following is a free-form translation of
;
; G -> H | H G
; H -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
;
; This routine checks to see if there is at least one digit. It fails if there
; isn't at least one digit; it succeeds and skips over all digits if there are
; one or more digits.

G               proc    near
                cmp     byte ptr es:[di], '0'       ;Check for at least
                jb      G_Failed                    ; one digit.
                cmp     byte ptr es:[di], '9'
                ja      G_Failed

DigitLoop:      inc     di                          ;Skip any remaining
                cmp     byte ptr es:[di], '0'       ; digits found.
                jb      G_Succeeds
                cmp     byte ptr es:[di], '9'
                jbe     DigitLoop
G_Succeeds:     stc
                ret

G_Failed:       clc                                 ;Fail if no digits
                ret                                 ; at all.
G               endp


; This main program tests the matching functions above and demonstrates
; how to call the matching functions.

Main            proc
                mov     ax, seg dseg ;Set up the segment registers
                mov     ds, ax
                mov     es, ax

                printf
                byte    "Enter an arithmetic expression: ",0
                lesi    InputLine
                gets
                call    E
                jnc     BadExp

; Good so far, but are we at the end of the string?

                cmp     byte ptr es:[di], 0
                jne     BadExp

; Okay, it truly is a good expression at this point.

                printf
```

```
                        byte    "'%s' is a valid expression",cr,lf,0
                        dword   InputLine
                        jmp     Quit

BadExp:                 printf
                        byte    "'%s' is an invalid arithmetic expression",cr,lf,0
                        dword   InputLine

Quit:                   ExitPgm
Main                    endp

cseg                    ends

sseg                    segment para stack 'stack'
stk                     byte    1024 dup ("stack ")
sseg                    ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               byte    16 dup (?)
zzzzzzseg               ends
                        end     Main
```

16.1.7 Some Final Comments on CFGs

The techniques presented in this chapter for converting CFGs to assembly code do not work for all CFGs. They only work for a (large) subset of the CFGs known as LL(1) grammars. The code that these techniques produce is a *recursive descent predictive parser*[6]. Although the set of context free languages recognizable by an LL(1) grammar is a subset of the context free languages, it is a very large subset and you shouldn't run into too many difficulties using this technique.

One important feature of predictive parsers is that they do not require any backtracking. If you are willing to live with the inefficiencies associated with backtracking, it is easy to extended a recursive descent parser to handle any CFG. Note that when you use backtracking, the *predictive* adjective goes away, you wind up with a nondeterministic system rather than a deterministic system (predictive and deterministic are very close in meaning in this case).

There are other CFG systems as well as LL(1). The so-called operator precedence and LR(k) CFGs are two examples. For more information about parsing and grammars, consult a good text on formal language theory or compiler construction (see the bibliography).

16.1.8 Beyond Context Free Languages

Although most patterns you will probably want to process will be regular or context free, there may be times when you need to recognize certain types of patterns that are beyond these two (e.g., *context sensitive* languages). As it turns out, the finite state automata are the simplest machines; the pushdown automata (that recognize context free languages) are the next step up. After pushdown automata, the next step up in power is the *Turing machine*. However, Turing machines are equivalent in power to the 80x86[7], so matching patterns recognized by Turing machines is no different than writing a normal program.

The key to writing functions that recognize patterns that are not context free is to maintain information in variables and use the variables to decide which of several productions you want to use at any one given time. This technique introduces *context sensitiv-*

6. A *parser* is a function that determines whether a pattern belongs to a language.

7. Actually, they are more powerful, in theory, because they have an infinite amount of memory available.

ity. Such techniques are very useful in artificial intelligence programs (like natural language processing) where ambiguity resolution depends on past knowledge or the current context of a pattern matching operation. However, the uses for such types of pattern matching quickly go beyond the scope of a text on assembly language programming, so we will let some other text continue this discussion.

16.2 The UCR Standard Library Pattern Matching Routines

The UCR Standard Library provides a very sophisticated set of pattern matching routines. They are patterned after the pattern matching facilities of SNOBOL4, support CFGs, and provide fully automatic backtracking, as necessary. Furthermore, by writing only *five* assembly language statements, you can match simple or complex patterns.

There is very little assembly language code to worry about when using the Standard Library's pattern matching routines because most of the work occurs in the data segment. To use the pattern matching routines, you first construct a pattern data structure in the data segment. You then pass the address of this pattern and the string you wish to test to the Standard Library match routine. The match routine returns failure or success depending on the state of the comparison. This isn't quite as easy as it sounds, though; learning how to construct the pattern data structure is almost like learning a new programming language. Fortunately, if you've followed the discussion on context free languages, learning this new "language" is a breeze.

The Standard Library *pattern* data structure takes the following form:

```
Pattern         struct
MatchFunction   dword   ?
MatchParm       dword   ?
MatchAlt        dword   ?
NextPattern     dword   ?
EndPattern      word    ?
StartPattern    word    ?
StrSeg          word    ?
Pattern         ends
```

The MatchFunction field contains the address of a routine to call to perform some sort of comparison. The success or failure of this function determines whether the pattern matches the input string. For example, the UCR Standard Library provides a MatchStr function that compares the next *n* characters of the input string against some other character string.

The MatchParm field contains the address or value of a parameter (if appropriate) for the MatchFunction routine. For example, if the MatchFunction routine is MatchStr, then the MatchParm field contains the address of the string to compare the input characters against. Likewise, the MatchChar routine compares the next input character in the string against the L.O. byte of the MatchParm field. Some matching functions do not require any parameters, they will ignore any value you assign to MatchParm field. By convention, most programmers store a zero in unused fields of the Pattern structure.

The MatchAlt field contains either zero (NULL) or the address of some other pattern data structure. If the current pattern matches the input characters, the pattern matching routines ignore this field. However, if the current pattern fails to match the input string, then the pattern matching routines will attempt to match the pattern whose address appears in this field. If this alternate pattern returns success, then the pattern matching routine returns success to the caller, otherwise it returns failure. If the MatchAlt field contains NULL, then the pattern matching routine immediately fails if the main pattern does not match.

The Pattern data structure only matches one item. For example, it might match a single character, a single string, or a character from a set of characters. A real world pattern will probably contain several small patterns concatenated together, e.g., the pattern for a Pascal identifier consists of a single character from the set of alphabetic characters followed

by one or more characters from the set [a-zA-Z0-9_]. The NextPattern field lets you create a composite pattern as the concatenation of two individual patterns. For such a composite pattern to return success, the current pattern must match and then the pattern specified by the NextPattern field must also match. Note that you can chain as many patterns together as you please using this field.

The last three fields, EndPattern, StartPattern, and StrSeg are for the internal use of the pattern matching routine. You should not modify or examine these fields.

Once you create a pattern, it is very easy to test a string to see if it matches that pattern. The calling sequence for the UCR Standard Library match routine is

```
lesi     « Input string to match »
ldxi     « Pattern to match string against »
mov      cx, 0
match
jc       Success
```

The Standard Library match routine expects a pointer to the input string in the es:di registers; it expects a pointer to the pattern you want to match in the dx:si register pair. The cx register should contain the length of the string you want to test. If cx contains zero, the match routine will test the entire input string. If cx contains a nonzero value, the match routine will only test the first cx characters in the string. Note that the end of the string (the zero terminating byte) must not appear in the string before the position specified in cx. For most applications, loading cx with zero before calling match is the most appropriate operation.

On return from the match routine, the carry flag denotes success or failure. If the carry flag is set, the pattern matches the string; if the carry flag is clear, the pattern does not match the string. Unlike the examples given in earlier sections, the match routine does not modify the di register, even if the match succeeds. Instead, it returns the failure/success position in the ax register. The is the position of the first character after the match if match succeeds, it is the position of the first unmatched character if match fails.

16.3 The Standard Library Pattern Matching Functions

The UCR Standard Library provides about 20 built-in pattern matching functions. These functions are based on the pattern matching facilities provided by the SNOBOL4 programming language, so they are very powerful indeed! You will probably discover that these routines solve all your pattern matching need, although it is easy to write your own pattern matching routines (see “Designing Your Own Pattern Matching Routines” on page 922) if an appropriate one is not available. The following subsections describe each of these pattern matching routines in detail.

There are two things you should note if you’re using the Standard Library’s SHELL.ASM file when creating programs that use pattern matching and character sets. First, there is a line at the very beginning of the SHELL.ASM file that contains the statement “matchfuncs”. This line is currently a comment because it contains a semicolon in column one. If you are going to be using the pattern matching facilities of the UCR Standard Library, you need to uncomment this line by deleting the semicolon in column one. If you are going to be using the character set facilities of the UCR Standard Library (very common when using the pattern matching facilities), you may want to uncomment the line containing “include stdsets.a” in the data segment. The “stdsets.a” file includes several common character sets, including alphabetics, digits, alphanumerics, whitespace, and so on.

16.3.1 Spancset

The spancset routine skips over all characters belonging to a character set. This routine will match zero or more characters in the specified set and, therefore, *always* succeeds.

The MatchParm field of the pattern data structure must point at a UCR Standard Library character set variable (see "The Character Set Routines in the UCR Standard Library" on page 856).

Example:

```
SkipAlphas      pattern  {spancset, alpha}
                  .
                  .
                  .
                lesi     StringWAlphas
                ldxi     SkipAlphas
                xor      cx, cx
                match
```

16.3.2 Brkcset

Brkcset is the *dual* to spancset – it matches zero or more characters in the input string which are *not* members of a specified character set. Another way of viewing brkcset is that it will match all characters in the input string *up to* a character in the specified character set (or to the end of the string). The matchparm field contains the address of the character set to match.

Example:

```
DoDigits        pattern  {brkcset, digits, 0, DoDigits2}
DoDigits2       pattern  {spancset, digits}
                  .
                  .
                  .
                lesi     StringWDigits
                ldxi     DoDigits
                xor      cx, cx
                match
                jnc      NoDigits
```

The code above matches any string that contains a string of one or more digits somewhere in the string.

16.3.3 Anycset

Anycset matches a single character in the input string from a set of characters. The matchparm field contains the address of a character set variable. If the next character in the input string is a member of this set, anycset set accepts the string and skips over than character. If the next input character is not a member of that set, anycset returns failure.

Example:

```
DoID            pattern  {anycset, alpha, 0, DoID2}
DoID2           pattern  {spancset, alphanum}
                  .
                  .
                  .
                lesi     StringWID
                ldxi     DoID
                xor      cx, cx
                match
                jnc      NoID
```

This code segment checks the string StringWID to see if it begins with an identifier specified by the regular expression [a-zA-Z][a-zA-Z0-9]*. The first subpattern with anycset makes sure there is an alphabetic character at the beginning of the string (alpha is the stdsets.a set variable that has all the alphabetic characters as members). If the string does not begin with an alphabetic, the DoID pattern fails. The second subpattern, DoID2, skips over any following alphanumeric characters using the spancset matching function. Note that spancset always succeeds.

The above code does *not* simply match a string that is an identifier; it matches strings that *begin* with a valid identifier. For example, it would match "ThisIsAnID" as well as "ThisIsAnID+SoIsThis - 5". If you only want to match a single identifier and nothing else, you must explicitly check for the end of string in your pattern. For more details on how to do this, see "EOS" on page 919.

16.3.4 Notanycset

Notanycset provides the complement to anycset – it matches a single character in the input string that is *not* a member of a character set. The matchparm field, as usual, contains the address of the character set whose members must not appear as the next character in the input string. If notanycset successfully matches a character (that is, the next input character is not in the designated character set), the function skips the character and returns success; otherwise it returns failure.

Example:

```
DoSpecial       pattern  {notanycset, digits, 0, DoSpecial2}
DoSpecial2      pattern  {spancset, alphanum}
                  .
                  .
                  .
                lesi     StringWSpecial
                ldxi     DoSpecial
                xor      cx, cx
                match
                jnc      NoSpecial
```

This code is similar to the DoID pattern in the previous example. It matches a string containing any character except a digit and then matches a string of alphanumeric characters.

16.3.5 MatchStr

Matchstr compares the next set of input characters against a character string. The matchparm field contains the address of a zero terminated string to compare against. If matchstr succeeds, it returns the carry set and skips over the characters it matched; if it fails, it tries the alternate matching function or returns failure if there is no alternate.

Example:

```
DoString        pattern  {matchstr, MyStr}
MyStr           byte     "Match this!",0
                  .
                  .
                  .
                lesi     String
                ldxi     DoString
                xor      cx, cx
                match
                jnc      NotMatchThis
```

This sample code matches any string that begins with the characters "Match This!"

16.3.6 MatchiStr

Matchistr is like matchstr insofar as it compares the next several characters against a zero terminated string value. However, matchistr does a *case insensitive* comparison. During the comparison it converts the characters in the input string to upper case before comparing them to the characters that the matchparm field points at. Therefore, *the string pointed at by the matchparm field must contain uppercase wherever alphabetics appear.* If the matchparm string contains any lower case characters, the matchistr function will always fail.

Example:

```
DoString        pattern  {matchistr, MyStr}
MyStr           byte     “MATCH THIS!”,0
                  .
                  .
                  .
                lesi     String
                ldxi     DoString
                xor      cx, cx
                match
                jnc      NotMatchThis
```

This example is identical to the one in the previous section except it will match the characters “match this!” using any combination of upper and lower case characters.

16.3.7 MatchToStr

Matchtostr matches all characters in an input string up to and including the characters specified by the matchparm parameter. This routine succeeds if the specified string appears somewhere in the input string, it fails if the string does not appear in the input string. This pattern function is quite useful for locating a substring and ignoring everything that came before the substring.

Example:

```
DoString        pattern  {matchtostr, MyStr}
MyStr           byte     “Match this!”,0
                  .
                  .
                  .
                lesi     String
                ldxi     DoString
                xor      cx, cx
                match
                jnc      NotMatchThis
```

Like the previous two examples, this code segment matches the string “Match this!” However, it does not require that the input string (String) begin with “Match this!” Instead, it only requires that “Match this!” appear somewhere in the string.

16.3.8 MatchChar

The matchchar function matches a single character. The matchparm field’s L.O. byte contains the character you want to match. If the next character in the input string is that character, then this function succeeds, otherwise it fails.

Example:

```
DoSpace         pattern  {matchchar, ‘ ‘}
                  .
                  .
                  .
                lesi     String
                ldxi     DoSpace
                xor      cx, cx
                match
                jnc      NoSpace
```

This code segment matches any string that begins with a space. Keep in mind that the match routine only checks the prefix of a string. If you wanted to see if the string contained only a space (rather than a string that begins with a space), you would need to explicitly check for an end of string after the space. Of course, it would be far more efficient to use strcmp (see “Strcmp, Strcmpl, Stricmp, Stricmpl” on page 848) rather than match for this purpose!

Note that unlike matchstr, you encode the character you want to match directly into the matchparm field. This lets you specify the character you want to test directly in the pattern definition.

16.3.9 MatchToChar

Like matchtostr, matchtochar matches all characters up to and including a character you specify. This is similar to brkcset except you don't have to create a character set containing a single member and brkcset skips up to *but not including* the specified character(s). Matchtochar fails if it cannot find the specified character in the input string.

Example:

```
DoToSpace       pattern  {matchtochar, ' '}
                  .
                  .
                  .
                lesi     String
                ldxi     DoSpace
                xor      cx, cx
                match
                jnc      NoSpace
```

This call to match will fail if there are no spaces left in the input string. If there are, the call to matchtochar will skip over all characters up to, and including, the first space. This is a useful pattern for skipping over words in a string.

16.3.10 MatchChars

Matchchars skips zero or more occurrences of a singe character in an input string. It is similar to spancset except you can specify a single character rather than an entire character set with a single member. Like matchchar, matchchars expects a single character in the L.O. byte of the matchparm field. Since this routine matches zero or more occurrences of that character, it always succeeds.

Example:

```
Skip2NextWord   pattern  {matchtochar, ' ', 0, SkipSpcs}
SkipSpcs        pattern  {matchchars, ' '}
                  .
                  .
                  .
                lesi     String
                ldxi     Skip2NextWord
                xor      cx, cx
                match
                jnc      NoWord
```

The code segment skips to the beginning of the next word in a string. It fails if there are no additional words in the string (i.e., the string contains no spaces).

16.3.11 MatchToPat

Matchtopat matches all characters in a string up to and including the substring matched by some other pattern. This is one of the two facilities the UCR Standard Library pattern matching routines provide to allow the implementation of nonterminal function calls (also see "SL_Match2" on page 922). This matching function succeeds if it finds a string matching the specified pattern somewhere on the line. If it succeeds, it skips the characters through the last character matched by the pattern parameter. As you would expect, the matchparm field contains the address of the pattern to match.

Example:

```
; Assume there is a pattern “expression” that matches arithmetic
; expressions. The following pattern determines if there is such an
; expression on the line followed by a semicolon.

FindExp         pattern  {matchtopat, expression, 0, MatchSemi}
MatchSemi       pattern  {matchchar, ‘;‘}
                  .
                  .
                  .
                lesi     String
                ldxi     FindExp
                xor      cx, cx
                match
                jnc      NoExp
```

16.3.12 EOS

The EOS pattern matches the end of a string. This pattern, which must obviously appear at the end of a pattern list if it appears at all, checks for the zero terminating byte. Since the Standard Library routines only match prefixes, you should stick this pattern at the end of a list if you want to ensure that a pattern exactly matches a string with no left over characters at the end. EOS succeeds if it matches the zero terminating byte, it fails otherwise.

Example:

```
SkipNumber      pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits, 0, EOSPat}
EOSPat          pattern  {EOS}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      NoNumber
```

The SkipNumber pattern matches strings that contain only decimal digits (from the start of the match to the end of the string). Note that EOS requires no parameters, not even a matchparm parameter.

16.3.13 ARB

ARB matches any number of arbitrary characters. This pattern matching function is equivalent to Σ^*. Note that ARB is a very inefficient routine to use. It works by assuming it can match all remaining characters in the string and then tries to match the pattern specified by the nextpattern field[8]. If the nextpattern item fails, ARB backs up one character and tries matching nextpattern again. This continues until the pattern specified by nextpattern succeeds or ARB backs up to its initial starting position. ARB succeeds if the pattern specified by nextpattern succeeds, it fails if it backs up to its initial starting position.

Given the enormous amount of backtracking that can occur with ARB (especially on long strings), you should try to avoid using this pattern if at all possible. The matchtostr, matchtochar, and matchtopat functions accomplish much of what ARB accomplishes, but they work forward rather than backward in the source string and may be more efficient. ARB is useful mainly if you’re sure the following pattern appears late in the string you’re matching or if the string you want to match occurs several times and you want to match the *last* occurrence (matchtostr, matchtochar, and matchtopat always match the first occurrence they find).

8. Since the match routine only matches prefixes, it does not make sense to apply ARB to the end of a pattern list, the same pattern would match with or without the final ARB. Therefore, ARB usually has a nextpattern field.

Example:

```
SkipNumber      pattern  {ARB,0,0,SkipDigit}
SkipDigit       pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      NoNumber
```

This code example matches the *last* number that appears on an input line. Note that ARB does not use the matchparm field, so you should set it to zero by default.

16.3.14 ARBNUM

ARBNUM matches an arbitrary number (zero or more) of patterns that occur in the input string. If *R* represents some nonterminal number (pattern matching function), then ARBNUM(*R*) is equivalent to the production $ARBNUM \rightarrow R\ ARBNUM \mid \varepsilon$.

The matchparm field contains the address of the pattern that ARBNUM attempts to match.

Example:

```
SkipNumbers     pattern  {ARBNUM, SkipNumber}
SkipNumber      pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits, 0, EndDigits}
EndDigits       pattern  {matchchars, ' ', EndString}
EndString       pattern  {EOS}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumbers
                xor      cx, cx
                match
                jnc      IllegalNumbers
```

This code accepts the input string if it consists of a sequence of zero or more numbers separated by spaces and terminated with the EOS pattern. Note the use of the matchalt field in the EndDigits pattern to select EOS rather than a space for the last number in the string.

16.3.15 Skip

Skip matches *n* arbitrary characters in the input string. The matchparm field is an integer value containing the number of characters to skip. Although the matchparm field is a double word, this routine limits the number of characters you can skip to 16 bits (65,535 characters); that is, *n* is the L.O. word of the matchparm field. This should prove sufficient for most needs.

Skip succeeds if there are at least *n* characters left in the input string; it fails if there are fewer than *n* characters left in the input string.

Example:

```
Skip1st6        pattern  {skip, 6, 0, SkipNumber}
SkipNumber      pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits, 0, EndDigits}
EndDigits       pattern  {EOS}
                  .
                  .
                  .
                lesi     String
                ldxi     Skip1st6
                xor      cx, cx
```

```
                match
                jnc     IllegalItem
```

This example matches a string containing six arbitrary characters followed by one or more decimal digits and a zero terminating byte.

16.3.16 Pos

Pos succeeds if the matching functions are currently at the nth character in the string, where *n* is the value in the L.O. word of the matchparm field. Pos fails if the matching functions are not currently at position *n* in the string. Unlike the pattern matching functions you've seen so far, pos does not consume any input characters. Note that the string starts out at position zero. So when you use the pos function, it succeeds if you've matched *n* characters at that point.

Example:

```
SkipNumber      pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits, 0, EndDigits}
EndDigits       pattern  {pos, 4}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      IllegalItem
```

This code matches a string that begins with exactly 4 decimal digits.

16.3.17 RPos

Rpos works quite a bit like the pos function except it succeeds if the current position is *n* character positions from the *end* of the string. Like pos, *n* is the L.O. 16 bits of the matchparm field. Also like pos, rpos does not consume any input characters.

Example:

```
SkipNumber      pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits, 0, EndDigits}
EndDigits       pattern  {rpos, 4}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      IllegalItem
```

This code matches any string that is all decimal digits except for the last four characters of the string. The string must be at least five characters long for the above pattern match to succeed.

16.3.18 GotoPos

Gotopos skips over any characters in the string until it reaches character position *n* in the string. This function fails if the pattern is already beyond position *n* in the string. The L.O. word of the matchparm field contains the value for *n*.

Example:

```
SkipNumber      pattern  {gotopos, 10, 0, MatchNmbr}
MatchNmbr       pattern  {anycset, digits, 0, SkipDigits}
```

```
SkipDigits      pattern  {spancset, digits, 0, EndDigits}
EndDigits       pattern  {rpos, 4}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      IllegalItem
```

This example code skips to position 10 in the string and attempts to match a string of digits starting with the 11th character. This pattern succeeds if the there are four characters remaining in the string after processing all the digits.

16.3.19 RGotoPos

Rgotopos works like gotopos except it goes to the position specified from the end of the string. Rgotopos fails if the matching routines are already beyond position *n* from the end of the string. As with gotopos, the L.O. word of the matchparm field contains the value for *n*.

Example:

```
SkipNumber      pattern  {rgotopos, 10, 0, MatchNmbr}
MatchNmbr       pattern  {anycset, digits, 0, SkipDigits}
SkipDigits      pattern  {spancset, digits}
                  .
                  .
                  .
                lesi     String
                ldxi     SkipNumber
                xor      cx, cx
                match
                jnc      IllegalItem
```

This example skips to ten characters from the end of the string and then attempts to match one or digits starting at that point. It fails if there aren't at least 11 characters in the string or the last 10 characters don't begin with a string of one or more digits.

16.3.20 SL_Match2

The sl_match2 routine is nothing more than a recursive call to match. The matchparm field contains the address of pattern to match. This is quite useful for simulating parenthesis around a pattern in a pattern expression. As far as matching strings are concerned, pattern1 and pattern2, below, are equivalent:

```
Pattern2        pattern  {sl_match2, Pattern1}
Pattern1        pattern  {matchchar, 'a'}
```

The only difference between invoking a pattern directly and invoking it with sl_match2 is that sl_match2 tweaks some internal variables to keep track of matching positions within the input string. Later, you can extract the character string matched by sl_match2 using the patgrab routine (see "Extracting Substrings from Matched Patterns" on page 925).

16.4 Designing Your Own Pattern Matching Routines

Although the UCR Standard Library provides a wide variety of matching functions, there is no way to anticipate the needs of all applications. Therefore, you will probably discover that the library does not support some particular pattern matching function you need. Fortunately, it is very easy for you to create your own pattern matching functions to augment those available in the UCR Standard Library. When you specify a matching func-

tion name in the pattern data structure, the match routine calls the specified address using a far call and passing the following parameters:

es:di- Points at the next character in the input string. You should not look at any characters before this address. Furthermore, you should never look beyond the end of the string (see cx below).

ds:si- Contains the four byte parameter found in the matchparm field.

cx- Contains the last position, plus one, in the input string you're allowed to look at. Note that your pattern matching routine should not look beyond location es:cx or the zero terminating byte; whichever comes first in the input string.

On return from the function, ax must contain the offset into the string (di's value) of the last character matched *plus one*, if your matching function is successful. It must also set the carry flag to denote success. After your pattern matches, the match routine might call another matching function (the one specified by the next pattern field) and that function begins matching at location es:ax.

If the pattern match fails, then you must return the original di value in the ax register and return with the carry flag clear. Note that your matching function must preserve all other registers.

There is one very important detail you must never forget with writing your own pattern matching routines – ds does not point at your data segment, it contains the H.O. word of the matchparm parameter. Therefore, if you are going to access global variables in your data segment you will need to push ds, load it with the address of dseg, and pop ds before leaving. Several examples throughout this chapter demonstrate how to do this.

There are some obvious omissions from (the current version of) the UCR Standard Library's repertoire. For example, there should probably be matchtoistr, matchichar, and matchtoichar pattern functions. The following example code demonstrates how to add a matchtoistr (match up to a string, doing a case insensitive comparison) routine.

```
                .xlist

                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list

dseg            segment   para public 'data'

TestString      byte      "This is the string 'xyz' in it",cr,lf,0

TestPat         pattern   {matchtoistr,xyz}
xyz             byte      "XYZ",0

dseg            ends


cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg

; MatchToiStr-  Matches all characters in a string up to, and including, the
;               specified parameter string. The parameter string must be
;               all upper case characters. This guy matches string using
;               a case insensitive comparison.
;
; inputs:
;               es:di-    Source string
;               ds:si-    String to match
;               cx-       Maximum match position
;
; outputs:
;               ax-       Points at first character beyond the end of the
;                         matched string if success, contains the initial DI
;                         value if failure occurs.
;               carry-    0 if failure, 1 if success.
```

```
MatchToiStr     proc    far
                pushf
                push    di
                push    si
                cld

; Check to see if we're already past the point were we're allowed
; to scan in the input string.

                cmp     di, cx
                jae     MTiSFailure

; If the pattern string is the empty string, always match.

                cmp     byte ptr ds:[si], 0
                je      MTSsuccess

; The following loop scans through the input string looking for
; the first character in the pattern string.

ScanLoop:       push    si
                lodsb               ;Get first char of string

                dec     di
FindFirst:      inc     di          ;Move on to next (or 1st) char.
                cmp     di, cx      ;If at cx, then we've got to
                jae CantFind1st; fail.

                mov     ah, es:[di] ;Get input character.
                cmp     ah, 'a'     ;Convert input character to
                jb      DoCmp       ; upper case if it's a lower
                cmp     ah, 'z'     ; case character.
                ja      DoCmp
                and     ah, 5fh
DoCmp:          cmp     al, ah      ;Compare input character against
                jne     FindFirst   ; pattern string.


; At this point, we've located the first character in the input string
; that matches the first character of the pattern string. See if the
; strings are equal.

                push    di          ;Save restart point.

CmpLoop:        cmp     di, cx      ;See if we've gone beyond the
                jae     StrNotThere; last position allowable.
                lodsb               ;Get next input character.
                cmp     al, 0       ;At the end of the parameter
                je      MTSsuccess2; string? If so, succeed.

                inc     di
                mov     ah, es:[di] ;Get the next input character.
                cmp     ah, 'a'     ;Convert input character to
                jb      DoCmp2      ; upper case if it's a lower
                cmp     ah, 'z'     ; case character.
                ja      DoCmp2
                and     ah, 5fh
DoCmp2:         cmp     al, ah      ;Compare input character against
                je      CmpLoop
                pop     di
                pop     si
                jmp     ScanLoop

StrNotThere:    add     sp, 2       ;Remove di from stack.
CantFind1st:    add     sp, 2       ;Remove si from stack.
MTiSFailure:    pop     si
                pop     di
                mov     ax, di      ;Return failure position in AX.
                popf
```

```
                clc                     ;Return failure.
                ret

MTSSuccess2:    add     sp, 2           ;Remove DI value from stack.
MTSSuccess:     add     sp, 2           ;Remove SI value from stack.
                mov     ax, di          ;Return next position in AX.
                pop     si
                pop     di
                popf
                stc                     ;Return success.
                ret
MatchToiStr     endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                lesi    TestString
                ldxi    TestPat
                xor     cx, cx
                match
                jnc     NoMatch
                print
                byte    "Matched",cr,lf,0
                jmp     Quit

NoMatch:        print
                byte    "Did not match",cr,lf,0

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

16.5 Extracting Substrings from Matched Patterns

Often, simply determining that a string matches a given pattern is insufficient. You may want to perform various operations that depend upon the actual information in that string. However, the pattern matching facilities described thus far do not provide a mechanism for testing individual components of the input string. In this section, you will see how to extract portions of a pattern for further processing.

Perhaps an example may help clarify the need to extract portions of a string. Suppose you are writing a stock buy/sell program and you want it to process commands described by the following regular expression:

```
(buy | sell) [0-9]+ shares of (ibm | apple | hp | dec)
```

While it is easy to devise a Standard Library pattern that recognizes strings of this form, calling the match routine would only tell you that you have a legal buy or sell command. It does not tell you if you are to buy or sell, *who* to buy or sell, or how many shares to buy or sell. Of course, you could take the cross product of (buy | sell) with (ibm | apple | hp | dec) and generate eight different regular expressions that uniquely determine whether you're buying or selling and whose stock you're trading, but you can't process the integer values this way (unless you willing to have *millions* of regular expressions). A better solu-

tion would be to extract substrings from the legal pattern and process these substrings after you verify that you have a legal buy or sell command. For example, you could extract buy or sell into one string, the digits into another, and the company name into a third. After verifying the syntax of the command, you could process the individual strings you've extracted. The UCR Standard Library patgrab routine provides this capability for you.

You normally call patgrab *after* calling match and verifying that it matches the input string. Patgrab expects a single parameter – a pointer to a pattern recently processed by match. Patgrab creates a string on the heap consisting of the characters matched by the given pattern and returns a pointer to this string in es:di. Note that patgrab only returns a string associated with a single pattern data structure, not a chain of pattern data structures. Consider the following pattern:

```
PatToGrab       pattern  {matchstr, str1, 0, Pat2}
Pat2            pattern  {matchstr, str2}
str1            byte     "Hello",0
str2            byte     " there",0
```

Calling match on PatToGrab will match the string "Hello there". However, if after calling match you call patgrab and pass it the address of PatToGrab, patgrab will return a pointer to the string "Hello".

Of course, you might want to collect a string that is the concatenation of several strings matched within your pattern (i.e., a portion of the pattern list). This is where calling the sl_match2 pattern matching function comes in handy. Consider the following pattern:

```
Numbers         pattern  {sl_match2, FirstNumber}
FirstNumber     pattern  {anycset, digits, 0, OtherDigs}
OtherDigs       pattern  {spancset, digits}
```

This pattern matches the same strings as

```
Numbers         pattern  {anycset, digits, 0, OtherDigs}
OtherDigs       pattern  {spancset, digits}
```

So why bother with the extra pattern that calls sl_match2? Well, as it turns out the sl_match2 matching function lets you create *parenthetical patterns*. A parenthetical pattern is a pattern list that the pattern matching routines (especially patgrab) treat as a single pattern. Although the match routine will match the same strings regardless of which version of Numbers you use, patgrab will produce two entirely different strings depending upon your choice of the above patterns. If you use the latter version, patgrab will only return the first digit of the number. If you use the former version (with the call to sl_match2), then patgrab returns the entire string matched by sl_match2, and that turns out to be the entire string of digits.

The following sample program demonstrates how to use parenthetical patterns to extract the pertinent information from the stock command presented earlier. It uses parenthetical patterns for the buy/sell command, the number of shares, and the company name.

```
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                matchfuncs
                .list

dseg            segment  para public 'data'

; Variables used to hold the number of shares bought/sold, a pointer to
; a string containing the buy/sell command, and a pointer to a string
; containing the company name.

Count           word     0
CmdPtr          dword    ?
CompPtr         dword    ?
```

```
; Some test strings to try out:

Cmd1            byte    “Buy 25 shares of apple stock”,0
Cmd2            byte    “Sell 50 shares of hp stock”,0
Cmd3            byte    “Buy 123 shares of dec stock”,0
Cmd4            byte    “Sell 15 shares of ibm stock”,0
BadCmd0         byte    “This is not a buy/sell command”,0

; Patterns for the stock buy/sell command:
;
; StkCmd matches buy or sell and creates a parenthetical pattern
; that contains the string “buy” or “sell”.

StkCmd          pattern {sl_match2, buyPat, 0, skipspcs1}

buyPat          pattern {matchistr,buystr,sellpat}
buystr          byte    “BUY”,0

sellpat         pattern {matchistr,sellstr}
sellstr         byte    “SELL”,0

; Skip zero or more white space characters after the buy command.

skipspcs1       pattern {spancset, whitespace, 0, CountPat}

; CountPat is a parenthetical pattern that matches one or more
; digits.

CountPat        pattern {sl_match2, Numbers, 0, skipspcs2}
Numbers         pattern {anycset, digits, 0, RestOfNum}
RestOfNum       pattern {spancset, digits}

; The following patterns match “ shares of “ allowing any amount
; of white space between the words.

skipspcs2       pattern {spancset, whitespace, 0, sharesPat}

sharesPat       pattern {matchistr, sharesStr, 0, skipspcs3}
sharesStr       byte    “SHARES”,0

skipspcs3       pattern {spancset, whitespace, 0, ofPat}

ofPat           pattern {matchistr, ofStr, 0, skipspcs4}
ofStr           byte    “OF”,0

skipspcs4       pattern {spancset, whitespace, 0, CompanyPat}

; The following parenthetical pattern matches a company name.
; The patgrab-available string will contain the corporate name.

CompanyPat      pattern {sl_match2, ibmpat}

ibmpat          pattern {matchistr, ibm, applePat}
ibm             byte    “IBM”,0

applePat        pattern {matchistr, apple, hpPat}
apple           byte    “APPLE”,0

hpPat           pattern {matchistr, hp, decPat}
hp              byte    “HP”,0

decPat          pattern {matchistr, decstr}
decstr          byte    “DEC”,0

                include stdsets.a
dseg            ends

cseg            segment para public ‘code’
                assume  cs:cseg, ds:dseg
```

```
; DoBuySell-     This routine processes a stock buy/sell command.
;               After matching the command, it grabs the components
;               of the command and outputs them as appropriate.
;               This routine demonstrates how to use patgrab to
;               extract substrings from a pattern string.
;
;               On entry, es:di must point at the buy/sell command
;               you want to process.

DoBuySell       proc    near
                ldxi StkCmd
                xor     cx, cx
                match
                jnc     NoMatch

                lesi    StkCmd
                patgrab
                mov     word ptr CmdPtr, di
                mov     word ptr CmdPtr+2, es

                lesi    CountPat
                patgrab
                atoi                ;Convert digits to integer
                mov     Count, ax
                free                ;Return storage to heap.

                lesi    CompanyPat
                patgrab
                mov     word ptr CompPtr, di
                mov     word ptr CompPtr+2, es

                printf
                byte    "Stock command: %^s\n"
                byte    "Number of shares: %d\n"
                byte    "Company to trade: %^s\n\n",0
                dword   CmdPtr, Count, CompPtr

                les     di, CmdPtr
                free
                les     di, CompPtr
                free
                ret

NoMatch:        print
                byte    "Illegal buy/sell command",cr,lf,0
                ret
DoBuySell       endp


Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit

                lesi    Cmd1
                call    DoBuySell
                lesi    Cmd2
                call    DoBuySell
                lesi    Cmd3
                call    DoBuySell
                lesi    Cmd4
                call    DoBuySell
                lesi    BadCmd0
                call    DoBuySell

Quit:           ExitPgm
Main            endp
```

```
cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

Sample program output:

```
Stock command: Buy
Number of shares: 25
Company to trade: apple

Stock command: Sell
Number of shares: 50
Company to trade: hp

Stock command: Buy
Number of shares: 123
Company to trade: dec

Stock command: Sell
Number of shares: 15
Company to trade: ibm

Illegal buy/sell command
```

16.6 Semantic Rules and Actions

Automata theory is mainly concerned with whether or not a string matches a given pattern. Like many theoretical sciences, practitioners of automata theory are only concerned if something is possible, the practical applications are not as important. For real programs, however, we would like to perform certain operations if we match a string or perform one from a set of operations depending on *how* we match the string.

A *semantic rule* or *semantic action* is an operation you perform based upon the type of pattern you match. This is, it is the piece of code you execute when you are satisfied with some pattern matching behavior. For example, the call to patgrab in the previous section is an example of a semantic action.

Normally, you execute the code associated with a semantic rule *after* returning from the call to match. Certainly when processing regular expressions, there is no need to process a semantic action in the *middle* of pattern matching operation. However, this isn't the case for a context free grammar. Context free grammars often involve recursion or may use the same pattern several times when matching a single string (that is, you may reference the same nonterminal several times while matching the pattern). The pattern matching data structure only maintains pointers (EndPattern, StartPattern, and StrSeg) to the last substring matched by a given pattern. Therefore, if you reuse a subpattern while matching a string and you need to execute a semantic rule associated with that subpattern, you will need to execute that semantic rule in the middle of the pattern matching operation, before you reference that subpattern again.

It turns out to be very easy to insert semantic rules in the middle of a pattern matching operation. All you need to do is write a pattern matching function that always succeeds (i.e., it returns with the carry flag clear). Within the body of your pattern matching routine you can choose to ignore the string the matching code is testing and perform any other actions you desire.

Your semantic action routine, on return, must set the carry flag and it must copy the original contents of di into ax. It must preserve all other registers. Your semantic action must *not* call the match routine (call sl_match2 instead). Match does not allow recursion (it is not *reentrant*) and calling match within a semantic action routine will mess up the pattern match in progress.

The following example provides several examples of semantic action routines within a program. This program converts arithmetic expressions in infix (algebraic) form to reverse polish notation (RPN) form.

```
; INFIX.ASM
;
; A simple program which demonstrates the pattern matching routines in the
; UCR library. This program accepts an arithmetic expression on the command
; line (no interleaving spaces in the expression is allowed, that is, there
; must be only one command line parameter) and converts it from infix notation
; to postfix (rpn) notation.

                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list


dseg            segment  para public 'data'

; Grammar for simple infix -> postfix translation operation
; (the semantic actions are enclosed in braces}:
;
; E -> FE'
; E' -> +F {output '+'} E' | -F {output '-'} E' | <empty string>
; F -> TF'
; F -> *T {output '*'} F' | /T {output '/'} F' | <empty string>
; T -> -T {output 'neg'} | S
; S -> <constant> {output constant} | (E)
;
; UCR Standard Library Pattern which handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

infix2rpn       pattern  {sl_Match2,E,,EndOfString}
EndOfString     pattern  {EOS}

; An "E" item consists of an "F" item optionally followed by "+" or "-"
; and another "E" item:

E               pattern  {sl_Match2, F,,Eprime}
Eprime          pattern  {MatchChar, '+', Eprime2, epf}
epf             pattern  {sl_Match2, F,,epPlus}
epPlus          pattern  {OutputPlus,,,Eprime}          ;Semantic rule

Eprime2         pattern  {MatchChar, '-', Succeed, emf}
emf             pattern  {sl_Match2, F,,epMinus}
epMinus         pattern  {OutputMinus,,,Eprime}         ;Semantic rule

; An "F" item consists of a "T" item optionally followed by "*" or "/"
; followed by another "T" item:

F               pattern  {sl_Match2, T,,Fprime}
Fprime          pattern  {MatchChar, '*', Fprime2, fmf}
fmf             pattern  {sl_Match2, T, 0, pMul}
pMul            pattern  {OutputMul,,,Fprime}           ;Semantic rule

Fprime2         pattern  {MatchChar, '/', Succeed, fdf}
fdf             pattern  {sl_Match2, T, 0, pDiv}
pDiv            pattern  {OutputDiv, 0, 0,Fprime}       ;Semantic rule
```

```
; T item consists of an “S” item or a “-” followed by another “T” item:

T               pattern  {MatchChar, ‘-’, S, TT}
TT              pattern  {sl_Match2, T, 0,tpn}
tpn             pattern  {OutputNeg}                    ;Semantic rule

; An “S” item is either a string of one or more digits or “(“ followed by
; and “E” item followed by “)”:

Const           pattern  {sl_Match2, DoDigits, 0, spd}
spd             pattern  {OutputDigits}                 ;Semantic rule

DoDigits        pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern  {Spancset, Digits}

S               pattern  {MatchChar, ‘(‘, Const, IntE}
IntE            pattern  {sl_Match2, E, 0, CloseParen}
CloseParen      pattern  {MatchChar, ‘)’}


Succeed         pattern  {DoSucceed}


                include  stdsets.a

dseg            ends



cseg            segment  para public ‘code’
                assume   cs:cseg, ds:dseg

; DoSucceed matches the empty string. In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed       proc     far
                mov      ax, di
                stc
                ret
DoSucceed       endp


; OutputPlus is a semantic rule which outputs the “+” operator after the
; parser sees a valid addition operator in the infix string.

OutputPlus      proc     far
                print
                byte     “ +”,0
                mov      ax, di                 ;Required by sl_Match
                stc
                ret
OutputPlus      endp


; OutputMinus is a semantic rule which outputs the “-” operator after the
; parser sees a valid subtraction operator in the infix string.

OutputMinus     proc     far
                print
                byte     “ -”,0
                mov      ax, di                 ;Required by sl_Match
                stc
                ret
OutputMinus     endp


; OutputMul is a semantic rule which outputs the “*” operator after the
; parser sees a valid multiplication operator in the infix string.
```

```
OutputMul       proc    far
                print
                byte    " *",0
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
OutputMul       endp


; OutputDiv is a semantic rule which outputs the "/" operator after the
; parser sees a valid division operator in the infix string.

OutputDiv       proc    far
                print
                byte    " /",0
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
OutputDiv       endp


; OutputNeg is a semantic rule which outputs the unary "-" operator after the
; parser sees a valid negation operator in the infix string.

OutputNeg       proc    far
                print
                byte    " neg",0
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
OutputNeg       endp


; OutputDigits outputs the numeric value when it encounters a legal integer
; value in the input string.

OutputDigits    proc    far
                push    es
                push    di
                mov     al, ' '
                putc
                lesi    const
                patgrab
                puts
                free
                stc
                pop     di
                mov     ax, di
                pop     es
                ret
OutputDigits    endp



; Okay, here's the main program which fetches the command line parameter
; and parses it.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit                         ; memory to the heap.


                print
                byte    "Enter an arithmetic expression: ",0
                getsm
                print
                byte    "Expression in postfix form: ",0
```

```
                ldxi    infix2rpn
                xor     cx, cx
                match
                jc      Succeeded

                print
                byte    “Syntax error”,0

Succeeded:      putcr

Quit:           ExitPgm
Main            endp

cseg            ends


; Allocate a reasonable amount of space for the stack (8k).

sseg            segment  para stack ‘stack’
stk             db       1024 dup (“stack “)
sseg            ends


; zzzzzzseg must be the last segment that gets loaded into memory!

zzzzzzseg       segment  para public ‘zzzzzz’
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

16.7 Constructing Patterns for the MATCH Routine

A major issue we have yet to discuss is how to convert regular expressions and context free grammars into patterns suitable for the UCR Standard Library pattern matching routines. Most of the examples appearing up to this point have used an ad hoc translation scheme; now it is time to provide an algorithm to accomplish this.

The following algorithm converts a context free grammar to a UCR Standard Library pattern data structure. If you want to convert a regular expression to a pattern, first convert the regular expression to a context free grammar (see “Converting REs to CFGs” on page 905). Of course, it is easy to convert many regular expression forms directly to a pattern, when such conversions are obvious you can bypass the following algorithm; for example, it should be obvious that you can use spancset to match a regular expression like [0-9]*.

The first step you must always take is to eliminate left recursion from the grammar. You will generate an infinite loop (and crash the machine) if you attempt to code a grammar containing left recursion into a pattern data structure. For information on eliminating left recursion, see “Eliminating Left Recursion and Left Factoring CFGs” on page 903. You might also want to left factor the grammar while you are eliminating left recursion. The Standard Library routines fully support backtracking, so left factoring is not strictly necessary, however, the matching routine will execute faster if it does not need to backtrack.

If a grammar production takes the form $A \rightarrow B\,C$ where A, B, and C are nonterminal symbols, you would create the following pattern:

```
A               pattern  {sl_match2,B,0,C}
```

This pattern description for A checks for an occurrence of a B pattern followed by a C pattern.

If *B* is a relatively simple production (that is, you can convert it to a single pattern data structure), you can optimize this to:

```
A               pattern  {B's Matching Function, B's parameter, 0, C}
```

The remaining examples will always call sl_match2, just to be consistent. However, as long as the nonterminals you invoke are simple, you can fold them into *A*"s pattern.

If a grammar production takes the form $A \rightarrow B \mid C$ where *A*, *B*, and *C* are nonterminal symbols, you would create the following pattern:

```
A               pattern  {sl_match2, B, C}
```

This pattern tries to match *B*. If it succeeds, *A* succeeds; if it fails, it tries to match *C*. At this point, *A*"s success or failure is the success or failure of *C*.

Handling terminal symbols is the next thing to consider. These are quite easy – all you need to do is use the appropriate matching function provided by the Standard Library, e.g., matchstr or matchchar. For example, if you have a production of the form $A \rightarrow$ abc | y you would convert this to the following pattern:

```
A               pattern  {matchstr,abc,ypat}
abc             byte     "abc",0
ypat            pattern  {matchchar,'y'}
```

The only remaining detail to consider is the empty string. If you have a production of the form $A \rightarrow \varepsilon$ then you need to write a pattern matching function that always succeed. The elegant way to do this is to write a custom pattern matching function. This function is

```
succeed       proc       far
              mov        ax, di            ;Required by sl_match
              stc                          ;Always succeed.
              ret
succeed       endp
```

Another, sneaky, way to force success is to use matchstr and pass it the empty string to match, e.g.,

```
success         pattern  {matchstr, emptystr}
emptystr        byte     0
```

The empty string always matches the input string, no matter what the input string contains.

If you have a production with several alternatives and ε is one of them, you must process ε last. For example, if you have the productions $A \rightarrow$ abc | y | *BC* | ε you would use the following pattern:

```
A               pattern  {matchstr,abc, tryY}
abc             byte     "abc",0
tryY            pattern  {matchchar, 'y', tryBC}
tryBC           pattern  {sl_match2, B, DoSuccess, C}
DoSuccess       pattern  {succeed}
```

While the technique described above will let you convert *any* CFG to a pattern that the Standard Library can process, it certainly does not take advantage of the Standard Library facilities, nor will it produce particularly efficient patterns. For example, consider the production:

Digits → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Converting this to a pattern using the techniques described above will yield the pattern:

```
Digits          pattern  {matchchar, '0', try1}
try1            pattern  {matchchar, '1', try2}
try2            pattern  {matchchar, '2', try3}
try3            pattern  {matchchar, '3', try4}
try4            pattern  {matchchar, '4', try5}
try5            pattern  {matchchar, '5', try6}
try6            pattern  {matchchar, '6', try7}
```

```
try7            pattern     {matchchar, '7', try8}
try8            pattern     {matchchar, '8', try9}
try9            pattern     {matchchar, '9'}
```

Obviously this isn't a very good solution because we can match this same pattern with the single statement:

```
Digits          pattern  {anycset, digits}
```

If your pattern is easy to specify using a regular expression, you should try to encode it using the built-in pattern matching functions and fall back on the above algorithm once you've handled the low level patterns as best you can. With experience, you will be able to choose an appropriate balance between the algorithm in this section and ad hoc methods you develop on your own.

16.8 Some Sample Pattern Matching Applications

The best way to learn how to convert a pattern matching problem to the respective pattern matching algorithms is by example. The following sections provide several examples of some small pattern matching problems and their solutions.

16.8.1 Converting Written Numbers to Integers

One interesting pattern matching problem is to convert written (English) numbers to their integer equivalents. For example, take the string "one hundred ninety-two" and convert it to the integer 192. Although written numbers represent a pattern quite a bit more complex than the ones we've seen thus far, a little study will show that it is easy to decompose such strings.

The first thing we will need to do is enumerate the English words we will need to process written numbers. This includes the following words:

> zero, one, two, three, four, five, six, seven, eight, nine, ten, eleven twelve, thirteen, fourteen, fifteen, sixteen, seventeen, eighteen, nineteen, twenty, thirty, forty, fifty sixty, seventy, eighty, ninety, hundred, *and* thousand.

With this set of words we can build all the values between zero and 65,535 (the values we can represent in a 16 bit integer.

Next, we've got to decide how to put these words together to form all the values between zero and 65,535. The first thing to note is that zero only occurs by itself, it is never part of another number. So our first production takes the form:

Number → zero | *NonZero*

The next thing to note is that certain values *may* occur in pairs, denoting addition. For example, eighty-five denotes the sum of eighty plus five. Also note that certain other pairs denote multiplication. If you have a statement like "two hundred" or "fifteen hundred" the "hundred" word says *multiply the preceding value by 100*. The multiplicative words, "hundred" and "thousand" , are also additive. Any value following these terms is added in to the total[9]; e.g., "one hundred five" means 1*100+5. By combining the appropriate rules, we obtain the following grammar

NonZero → *Thousands Maybe100s* | *Hundreds*
Thousands → *Under100* thousand
Maybe100s → *Hundreds* | ε
Hundreds → *Under100* hundred *After100* | *Under100*
After100 → *Under100* | ε

9. We will ignore special multiplicative forms like "one thousand thousand" (one million) because these forms are all too large to fit into 16 bits. .

```
Under100 →     Tens Maybe1s| Teens | ones
Maybe1s  →     Ones | ε
ones →   one | two | three | four | five | six | seven | eight | nine
teens →  ten | eleven | twelve | thirteen | fourteen | fifteen | sixteen |
         seventeen | eighteen | nineteen
tens →   twenty | thirty | forty | fifty | sixty | seventy | eighty | ninety
```

The final step is to add semantic actions to actually convert the strings matched by this grammar to integer values. The basic idea is to initialize an accumulator value to zero. Whenever you encounter one of the strings that *ones, teens,* or *tens* matches, you add the corresponding value to the accumulator. If you encounter the hundred or thousand strings, you multiply the accumulator by the appropriate factor. The complete program to do the conversion follows:

```
; Numbers.asm
;
; This program converts written English numbers in the range “zero”
; to “sixty five thousand five hundred thirty five” to the corresponding
; integer value.

                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list


dseg            segment  para public ‘data’

Value           word     0                      ;Store results here.
HundredsVal     word     0
ThousandsVal    word     0


Str0            byte     “twenty one”,0
Str1            byte     “nineteen hundred thirty-five”,0
Str2            byte     “thirty three thousand two hundred nineteen”,0
Str3            byte     “three”,0
Str4            byte     “fourteen”,0
Str5            byte     “fifty two”,0
Str6            byte     “seven hundred”,0
Str7            byte     “two thousand seven”,0
Str8            byte     “four thousand ninety six”,0
Str9            byte     “five hundred twelve”,0
Str10           byte     “twenty three thousand two hundred ninety-five”,0
Str11           byte     “seventy-five hundred”,0
Str12           byte     “sixty-five thousand”,0
Str13           byte     “one thousand”,0


; The following grammar is what we use to process the numbers.
; Semantic actions appear in the braces.
;
; Note: begin by initializing Value, HundredsVal, and ThousandsVal to zero.
;
; N             -> separators zero
;               | N4
;
; N4            -> do1000s maybe100s
;               | do100s
;
; Maybe100s     -> do100s
;               | <empty string>
;
; do1000s       -> Under100 “THOUSAND” separators
;                           {ThousandsVal := Value*1000}
;
; do100s        -> Under100 “HUNDRED”
```

```
;                       {HundredsVal := Value*100} After100
;               | Under100
;
; After100      -> {Value := 0} Under100
;               | {Value := 0} <empty string>
;
; Under100      -> {Value := 0} try20 try1s
;               | {Value := 0} doTeens
;               | {Value := 0} do1s
;
; try1s         -> do1s | <empty string>
;
; try20         -> “TWENTY” {Value := Value + 20}
;               | “THIRTY” {Value := Value + 30}
;               | ...
;               | “NINETY” {Value := Value + 90}
;
; doTeens       -> “TEN” {Value := Value + 10}
;               | “ELEVEN” {Value := Value + 11}
;               | ...
;               | “NINETEEN” {Value := Value + 19}
;
; do1s          -> “ONE” {Value := Value + 1}
;               | “TWO”  {Value := Value + 2}
;               | ...
;               | “NINE” {Value := Value + 9}


separators      pattern  {anycset, delimiters, 0, delim2}
delim2          pattern  {spancset, delimiters}
doSuccess       pattern  {succeed}
AtLast          pattern  {sl_match2, separators, AtEOS, AtEOS}
AtEOS           pattern  {EOS}


N               pattern  {sl_match2, separators, N2, N2}
N2              pattern  {matchistr, zero, N3, AtLast}
zero            byte     “ZERO”,0

N3              pattern  {sl_match2, N4, 0, AtLast}
N4              pattern  {sl_match2, do1000s, do100s, Maybe100s}
Maybe100s       pattern  {sl_match2, do100s, AtLast, AtLast}

do1000s         pattern  {sl_match2, Under100, 0, do1000s2}
do1000s2        pattern  {matchistr, str1000, 0, do1000s3}
do1000s3        pattern  {sl_match2, separators, do1000s4, do1000s5}
do1000s4        pattern  {EOS, 0, 0, do1000s5}
do1000s5        pattern  {Get1000s}
str1000         byte     “THOUSAND”,0

do100s          pattern  {sl_match2, do100s1, Under100, After100}
do100s1         pattern  {sl_match2, Under100, 0, do100s2}
do100s2         pattern  {matchistr, str100, 0, do100s3}
do100s3         pattern  {sl_match2, separators, do100s4, do100s5}
do100s4         pattern  {EOS, 0, 0, do100s5}
do100s5         pattern  {Get100s}
str100          byte     “HUNDRED”,0

After100        pattern  {SetVal, 0, 0, After100a}
After100a       pattern  {sl_match2, Under100, doSuccess}

Under100        pattern  {SetVal, 0, 0, Under100a}
Under100a       pattern  {sl_match2, try20, Under100b, Do1orE}
Under100b       pattern  {sl_match2, doTeens, do1s}

Do1orE          pattern  {sl_match2, do1s, doSuccess, 0}



NumPat          macro    lbl, next, Constant, string
```

```
                        local   try, SkipSpcs, val, str, tryEOS
lbl                     pattern {sl_match2, try, next}
try                     pattern {matchistr, str, 0, SkipSpcs}
SkipSpcs                pattern {sl_match2, separators, tryEOS, val}
tryEOS                  pattern {EOS, 0, 0, val}
val                     pattern {AddVal, Constant}
str                     byte    string
                        byte    0
                        endm

                        NumPat  doTeens, try11, 10, "TEN"
                        NumPat  try11, try12, 11, "ELEVEN"
                        NumPat  try12, try13, 12, "TWELVE"
                        NumPat  try13, try14, 13, "THIRTEEN"
                        NumPat  try14, try15, 14, "FOURTEEN"
                        NumPat  try15, try16, 15, "FIFTEEN"
                        NumPat  try16, try17, 16, "SIXTEEN"
                        NumPat  try17, try18, 17, "SEVENTEEN"
                        NumPat  try18, try19, 18, "EIGHTEEN"
                        NumPat  try19, 0, 19, "NINETEEN"

                        NumPat  do1s, try2, 1, "ONE"
                        NumPat  try2, try3, 2, "TWO"
                        NumPat  try3, try4, 3, "THREE"
                        NumPat  try4, try5, 4, "FOUR"
                        NumPat  try5, try6, 5, "FIVE"
                        NumPat  try6, try7, 6, "SIX"
                        NumPat  try7, try8, 7, "SEVEN"
                        NumPat  try8, try9, 8, "EIGHT"
                        NumPat  try9, 0, 9, "NINE"

                        NumPat  try20, try30, 20, "TWENTY"
                        NumPat  try30, try40, 30, "THIRTY"
                        NumPat  try40, try50, 40, "FORTY"
                        NumPat  try50, try60, 50, "FIFTY"
                        NumPat  try60, try70, 60, "SIXTY"
                        NumPat  try70, try80, 70, "SEVENTY"
                        NumPat  try80, try90, 80, "EIGHTY"
                        NumPat  try90, 0, 90, "NINETY"

                        include stdsets.a

dseg                    ends



cseg                    segment para public 'code'
                        assume  cs:cseg, ds:dseg


; Semantic actions for our grammar:
;
;
;
; Get1000s-     We've just processed the value one..nine, grab it from
;               the value variable, multiply it by 1000, and store it
;               into thousandsval.

Get1000s        proc    far
                push    ds
                push    dx
                mov     ax, dseg
                mov     ds, ax

                mov     ax, 1000
                mul     Value
                mov     ThousandsVal, ax
                mov     Value, 0

                pop     dx
```

```
                mov     ax, di                  ;Required by sl_match.
                pop     ds
                stc                             ;Always return success.
                ret
Get1000s        endp


; Get100s-      We've just processed the value one..nine, grab it from
;               the value variable, multiply it by 100, and store it
;               into hundredsval.

Get100s         proc    far
                push    ds
                push    dx
                mov     ax, dseg
                mov     ds, ax

                mov     ax, 100
                mul     Value
                mov     HundredsVal, ax
                mov     Value, 0

                pop     dx
                mov     ax, di                  ;Required by sl_match.
                pop     ds
                stc                             ;Always return success.
                ret
Get100s         endp


; SetVal-       This routine sets Value to whatever is in si

SetVal          proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     Value, si
                mov     ax, di
                pop     ds
                stc
                ret
SetVal          endp

; AddVal-       This routine sets adds whatever is in si to Value

AddVal          proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                add     Value, si
                mov     ax, di
                pop     ds
                stc
                ret
AddVal          endp


; Succeed matches the empty string. In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

Succeed         proc    far
                mov     ax, di
                stc
                ret
Succeed         endp


; This subroutine expects a pointer to a string containing the English
; version of an integer number. It converts this to an integer and
```

```
; prints the result.

ConvertNumber   proc    near
                mov     value, 0
                mov     HundredsVal, 0
                mov     ThousandsVal, 0

                ldxi    N
                xor     cx, cx
                match
                jnc     NoMatch
                mov     al, "'"
                putc
                puts
                print
                byte    "' = ", 0
                mov     ax, ThousandsVal
                add     ax, HundredsVal
                add     ax, Value
                putu
                putcr
                jmp     Done

NoMatch:        print
                byte    "Illegal number",cr,lf,0

Done:           ret
ConvertNumber   endp



Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit                         ;Init memory manager.

; Union in a "-" to the delimiters set because numbers can have
; dashes in them.

                lesi    delimiters
                mov     al, '-'
                addchar

; Some calls to test the ConvertNumber routine and the conversion process.

                lesi    Str0
                call    ConvertNumber
                lesi    Str1
                call    ConvertNumber
                lesi    Str2
                call    ConvertNumber
                lesi    Str3
                call    ConvertNumber
                lesi    Str4
                call    ConvertNumber
                lesi    Str5
                call    ConvertNumber
                lesi    Str6
                call    ConvertNumber
                lesi    Str7
                call    ConvertNumber
                lesi    Str8
                call    ConvertNumber
                lesi    Str9
                call    ConvertNumber
                lesi    Str10
                call    ConvertNumber
                lesi    Str11
```

```
                call    ConvertNumber
                lesi    Str12
                call    ConvertNumber
                lesi    Str13
                call    ConvertNumber


Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

Sample output:

```
'twenty one' = 21
'nineteen hundred thirty-five' = 1935
'thirty three thousand two hundred nineteen' = 33219
'three' = 3
'fourteen' = 14
'fifty two' = 52
'seven hundred' = 700
'two thousand seven' = 2007
'four thousand ninety six' = 4096
'five hundred twelve' = 512
'twenty three thousand two hundred ninety-five' = 23295
'seventy-five hundred' = 7500
'sixty-five thousand' = 65000
'one thousand' = 1000
```

16.8.2 Processing Dates

Another useful program that converts English text to numeric form is a date processor. A date processor takes strings like "Jan 23, 1997" and converts it to three integer values representing the month, day, and year. Of course, while we're at it, it's easy enough to modify the grammar for date strings to allow the input string to take any of the following common date formats:

```
Jan 23, 1997
January 23, 1997
23 Jan, 1997
23 January, 1997
1/23/97
1-23-97
1/23/1997
1-23-1997
```

In each of these cases the date processing routines should store one into the variable month, 23 into the variable day, and 1997 into the year variable (we will assume all years are in the range 1900-1999 if the string supplies only two digits for the year). Of course, we could also allow dates like "January twenty-third, nineteen hundred and ninety seven" by using an number processing parser similar to the one presented in the previous section. However, that is an exercise left to the reader.

The grammar to process dates is

```
Date→           EngMon Integer Integer |
                Integer EngMon Integer |
```

```
                Integer / Integer / Integer |
                Integer - Integer - Integer

EngMon →        JAN | JANUARY | FEB | FEBRUARY | … | DEC | DECEMBER
Integer →       digit Integer | digit
digit  →        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

We will use some semantic rules to place some restrictions on these strings. For example, the grammar above allows integers of any size; however, months must fall in the range 1-12 and days must fall in the range 1-28, 1-29, 1-30, or 1-31 depending on the year and month. Years must fall in the range 0-99 or 1900-1999.

Here is the 80x86 code for this grammar:

```
; datepat.asm
;
; This program converts dates of various formats to a three integer
; component value- month, day, and year.

                .xlist
                .286
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list
                .lall


dseg            segment  para public 'data'

; The following three variables hold the result of the conversion.

month           word     0
day             word     0
year            word     0

; StrPtr is a double word value that points at the string under test.
; The output routines use this variable. It is declared as two word
; values so it is easier to store es:di into it.

strptr          word     0,0

; Value is a generic variable the ConvertInt routine uses

value           word     0



; Number of valid days in each month (Feb is handled specially)

DaysInMonth     byte     31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31



; Some sample strings to test the date conversion routines.

Str0            byte     "Feb 4, 1956",0
Str1            byte     "July 20, 1960",0
Str2            byte     "Jul 8, 1964",0
Str3            byte     "1/1/97",0
Str4            byte     "1-1-1997",0
Str5            byte     "12-25-74",0
Str6            byte     "3/28/1981",0
Str7            byte     "January 1, 1999",0
Str8            byte     "Feb 29, 1996",0
Str9            byte     "30 June, 1990",0
Str10           byte     "August 7, 1945",0
Str11           byte     "30 September, 1992",0
Str12           byte     "Feb 29, 1990",0
Str13           byte     "29 Feb, 1992",0
```

```
; The following grammar is what we use to process the dates
;
; Date ->       EngMon Integer Integer
;      |        Integer EngMon Integer
;      |        Integer "/" Integer "/" Integer
;      |        Integer "-" Integer "-" Integer
;
; EngMon->      Jan | January | Feb | February | ... | Dec | December
; Integer->     digit integer | digit
; digit->       0 | 1 | ... | 9
;
; Some semantic rules this code has to check:
;
; If the year is in the range 0-99, this code has to add 1900 to it.
; If the year is not in the range 0-99 or 1900-1999 then return an error.
; The month must be in the range 1-12, else return an error.
; The day must be between one and 28, 29, 30, or 31. The exact maximum
; day depends on the month.


separators      pattern  {spancset, delimiters}


; DatePat processes dates of the form "MonInEnglish Day Year"

DatePat         pattern  {sl_match2, EngMon, DatePat2, DayYear}
DayYear         pattern  {sl_match2, DayInteger, 0, YearPat}
YearPat         pattern  {sl_match2, YearInteger}

; DatePat2 processes dates of the form "Day MonInEng Year"

DatePat2        pattern  {sl_match2, DayInteger, DatePat3, MonthYear}
MonthYear       pattern  {sl_match2, EngMon, 0, YearPat}

; DatePat3 processes dates of the form "mm-dd-yy"

DatePat3        pattern  {sl_match2, MonInteger, DatePat4, DatePat3a}
DatePat3a       pattern  {sl_match2, separators, DatePat3b, DatePat3b}
DatePat3b       pattern  {matchchar, '-', 0, DatePat3c}
DatePat3c       pattern  {sl_match2, DayInteger, 0, DatePat3d}
DatePat3d       pattern  {sl_match2, separators, DatePat3e, DatePat3e}
DatePat3e       pattern  {matchchar, '-', 0, DatePat3f}
DatePat3f       pattern  {sl_match2, YearInteger}

; DatePat4 processes dates of the form "mm/dd/yy"

DatePat4        pattern  {sl_match2, MonInteger, 0, DatePat4a}
DatePat4a       pattern  {sl_match2, separators, DatePat4b, DatePat4b}
DatePat4b       pattern  {matchchar, '/', 0, DatePat4c}
DatePat4c       pattern  {sl_match2, DayInteger, 0, DatePat4d}
DatePat4d       pattern  {sl_match2, separators, DatePat4e, DatePat4e}
DatePat4e       pattern  {matchchar, '/', 0, DatePat4f}
DatePat4f       pattern  {sl_match2, YearInteger}


; DayInteger matches an decimal string, converts it to an integer, and
; stores the result away in the Day variable.

DayInteger      pattern  {sl_match2, Integer, 0, SetDayPat}
SetDayPat       pattern  {SetDay}

; MonInteger matches an decimal string, converts it to an integer, and
; stores the result away in the Month variable.

MonInteger      pattern  {sl_match2, Integer, 0, SetMonPat}
SetMonPat       pattern  {SetMon}
```

```
; YearInteger matches an decimal string, converts it to an integer, and
; stores the result away in the Year variable.


YearInteger     pattern  {sl_match2, Integer, 0, SetYearPat}
SetYearPat      pattern  {SetYear}


; Integer skips any leading delimiter characters and then matches a
; decimal string. The Integer0 pattern matches exactly the decimal
; characters; the code does a patgrab on Integer0 when converting
; this string to an integer.

Integer         pattern  {sl_match2, separators, 0, Integer0}
Integer0        pattern  {sl_match2, number, 0, Convert2Int}
number          pattern  {anycset, digits, 0, number2}
number2         pattern  {spancset, digits}
Convert2Int     pattern  {ConvertInt}



; A macro to make it easy to declare each of the 24 English month
; patterns (24 because we allow the full month name and an
; abbreviation).

MoPat           macro    name, next, str, str2, value
                local SetMo, string, full, short, string2, doMon

name            pattern  {sl_match2, short, next}
short           pattern  {matchistr, string2, full, SetMo}
full            pattern  {matchistr, string, 0, SetMo}

string          byte str
                byte     0

string2         byte     str2
                byte     0

SetMo           pattern  {MonthVal, value}
                endm


; EngMon is a chain of patterns that match one of the strings
; JAN, JANUARY, FEB, FEBRUARY, etc. The last parameter to the
; MoPat macro is the month number.

EngMon          pattern {sl_match2, separators, jan, jan}
                MoPat    jan, feb, "JAN", "JANUARY", 1
                MoPat    feb, mar, "FEB", "FEBRUARY", 2
                MoPat    mar, apr, "MAR", "MARCH", 3
                MoPat    apr, may, "APR", "APRIL", 4
                MoPat    may, jun, "MAY", "MAY", 5
                MoPat    jun, jul, "JUN", "JUNE", 6
                MoPat    jul, aug, "JUL", "JULY", 7
                MoPat    aug, sep, "AUG", "AUGUST", 8
                MoPat    sep, oct, "SEP", "SEPTEMBER", 9
                MoPat    oct, nov, "OCT", "OCTOBER", 10
                MoPat    nov, decem, "NOV", "NOVEMBER", 11
                MoPat    decem, 0, "DEC", "DECEMBER", 12



; We use the "digits" and "delimiters" sets from the standard library.

                include  stdsets.a

dseg            ends
```

```
cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


; ConvertInt-    Matches a sequence of digits and converts them to an integer.

ConvertInt      proc     far
                push     ds
                push     es
                push     di
                mov      ax, dseg
                mov      ds, ax

                lesi Integer0         ;Integer0 contains the decimal
                patgrab               ; string we matched, grab that
                atou                  ; string and convert it to an
                mov      Value, ax    ; integer and save the result.
                free                  ;Free mem allocated by patgrab.

                pop      di
                mov      ax, di       ;Required by sl_match.
                pop      es
                pop      ds
                stc                   ;Always succeed.
                ret

ConvertInt      endp


; SetDay, SetMon, and SetYear simply copy value to the appropriate
; variable.

SetDay          proc     far
                push     ds
                mov      ax, dseg
                mov      ds, ax
                mov      ax, value
                mov      day, ax
                mov      ax, di
                pop      ds
                stc
                ret
SetDay          endp


SetMon          proc     far
                push     ds
                mov      ax, dseg
                mov      ds, ax
                mov      ax, value
                mov      Month, ax
                mov      ax, di
                pop      ds
                stc
                ret
SetMon          endp


SetYear         proc     far
                push     ds
                mov      ax, dseg
                mov      ds, ax
                mov      ax, value
                mov      Year, ax
                mov      ax, di
                pop      ds
                stc
                ret
```

```
SetYear         endp


; MonthVal is a pattern used by the English month patterns.
; This pattern function simply copies the matchparm field to
; the month variable (the matchparm field is passed in si).

MonthVal        proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     Month, si
                mov     ax, di
                pop     ds
                stc
                ret
MonthVal        endp



; ChkDate-      Checks a date to see if it is valid. Returns with the
;               carry flag set if it is, clear if not.

ChkDate         proc    far
                push    ds
                push    ax
                push    bx

                mov     ax, dseg
                mov     ds, ax

; If the year is in the range 0-99, add 1900 to it.
; Then check to see if it's in the range 1900-1999.

                cmp     Year, 100
                ja      Notb100
                add     Year, 1900
Notb100:        cmp     Year, 2000
                jae     BadDate
                cmp     Year, 1900
                jb      BadDate

; Okay, make sure the month is in the range 1-12

                cmp     Month, 12
                ja      BadDate
                cmp     Month, 1
                jb      BadDate

; See if the number of days is correct for all months except Feb:

                mov     bx, Month
                mov     ax, Day                 ;Make sure Day <> 0.
                test    ax, ax
                je      BadDate
                cmp     ah, 0                   ;Make sure Day < 256.
                jne     BadDate

                cmp     bx, 2                   ;Handle Feb elsewhere.
                je      DoFeb
                cmp     al, DaysInMonth[bx-1]   ;Check against max val.
                ja      BadDate
                jmp     GoodDate

; Kludge to handle leap years. Note that 1900 is *not* a leap year.

DoFeb:          cmp     ax, 29                  ;Only applies if day is
                jb      GoodDate                ; equal to 29.
                ja      BadDate                 ;Error if Day > 29.
                mov     bx, Year                ;1900 is not a leap year
```

```
                cmp     bx, 1900                ; so handle that here.
                je      BadDate
                and     bx, 11b                 ;Else, Year mod 4 is a
                jne     BadDate                 ; leap year.

GoodDate:       pop     bx
                pop     ax
                pop     ds
                stc
                ret

BadDate:        pop     bx
                pop     ax
                pop     ds
                clc
                ret
ChkDate         endp


; ConvertDate-  ES:DI contains a pointer to a string containing a valid
;               date. This routine converts that date to the three
;               integer values found in the Month, Day, and Year
;               variables. Then it prints them to verify the pattern
;               matching routine.

ConvertDate     proc    near

                ldxi    DatePat
                xor     cx, cx
                match
                jnc     NoMatch

                mov     strptr, di              ;Save string pointer for
                mov     strptr+2, es            ; use by printf

                call    ChkDate                 ;Validate the date.
                jnc     NoMatch

                printf
                byte    "%-20^s = Month: %2d Day: %2d Year: %4d\n",0
                dword   strptr, Month, Day, Year
                jmp     Done

NoMatch:        printf
                byte    "Illegal date ('%^s')",cr,lf,0
                dword   strptr

Done:           ret
ConvertDate     endp




Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                meminit                         ;Init memory manager.

; Call ConvertDate to test several different date strings.

                lesi    Str0
                call    ConvertDate
                lesi    Str1
                call    ConvertDate
                lesi    Str2
                call    ConvertDate
                lesi    Str3
                call    ConvertDate
```

```
                        lesi    Str4
                        call    ConvertDate
                        lesi    Str5
                        call    ConvertDate
                        lesi    Str6
                        call    ConvertDate
                        lesi    Str7
                        call    ConvertDate
                        lesi    Str8
                        call    ConvertDate
                        lesi    Str9
                        call    ConvertDate
                        lesi    Str10
                        call    ConvertDate
                        lesi    Str11
                        call    ConvertDate
                        lesi    Str12
                        call    ConvertDate
                        lesi    Str13
                        call    ConvertDate


Quit:                   ExitPgm
Main                    endp

cseg                    ends

sseg                    segment  para stack 'stack'
stk                     db       1024 dup ("stack ")
sseg                    ends

zzzzzzseg               segment  para public 'zzzzzz'
LastBytes               db       16 dup (?)
zzzzzzseg               ends
                        end      Main
```

Sample Output:

```
Feb 4, 1956             = Month:  2 Day:  4 Year: 1956
July 20, 1960           = Month:  7 Day: 20 Year: 1960
Jul 8, 1964             = Month:  7 Day:  8 Year: 1964
1/1/97                  = Month:  1 Day:  1 Year: 1997
1-1-1997                = Month:  1 Day:  1 Year: 1997
12-25-74                = Month: 12 Day: 25 Year: 1974
3/28/1981               = Month:  3 Day: 28 Year: 1981
January 1, 1999         = Month:  1 Day:  1 Year: 1999
Feb 29, 1996            = Month:  2 Day: 29 Year: 1996
30 June, 1990           = Month:  6 Day: 30 Year: 1990
August 7, 1945          = Month:  8 Day:  7 Year: 1945
30 September, 1992      = Month:  9 Day: 30 Year: 1992
Illegal date ('Feb 29, 1990')
29 Feb, 1992            = Month:  2 Day: 29 Year: 1992
```

16.8.3 Evaluating Arithmetic Expressions

Many programs (e.g., spreadsheets, interpreters, compilers, and assemblers) need to process arithmetic expressions. The following example provides a simple calculator that operates on floating point numbers. This particular program uses the 80x87 FPU chip, although it would not be too difficult to modify it so that it uses the floating point routines in the UCR Standard Library.

```
; ARITH2.ASM
;
; A simple floating point calculator that demonstrates the use of the
; UCR Standard Library pattern matching routines. Note that this
```

```
; program requires an FPU.

                .xlist
                .386
                .387
                option   segment:use16
                include    stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list

dseg            segment  para public 'data'

; The following is a temporary used when converting a floating point
; string to a 64 bit real value.

CurValue        real8    0.0

; Some sample strings containing expressions to try out:

Str1            byte     "5+2*(3-1)",0
Str2            byte     "(5+2)*(7-10)",0
Str3            byte     "5",0
Str4            byte     "(6+2)/(5+1)-7e5*2/1.3e2+1.5",0
Str5            byte     "2.5*(2-(3+1)/4+1)",0
Str6            byte     "6+(-5*2)",0
Str7            byte     "6*-1",0
Str8            byte     "1.2e5/2.1e5",0
Str9            byte     "0.9999999999999999+1e-15",0
str10           byte     "2.1-1.1",0

; Grammar for simple infix -> postfix translation operation:
; Semantic rules appear in braces.
;
; E -> FE' {print result}
; E' -> +F {fadd} E' | -F {fsub} E' | <empty string>
; F -> TF'
; F -> *T {fmul} F' | /T {fdiv} F' | <empty string>
; T -> -T {fchs} | S
; S -> <constant> {fld constant} | (E)
;
;
;
; UCR Standard Library Pattern which handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

Expression      pattern  {sl_Match2,E,,EndOfString}
EndOfString     pattern  {EOS}

; An "E" item consists of an "F" item optionally followed by "+" or "-"
; and another "E" item:

E               pattern  {sl_Match2, F,,Eprime}
Eprime          pattern  {MatchChar, '+', Eprime2, epf}
epf             pattern  {sl_Match2, F,,epPlus}
epPlus          pattern  {DoFadd,,,Eprime}

Eprime2         pattern  {MatchChar, '-', Succeed, emf}
emf             pattern  {sl_Match2, F,,epMinus}
epMinus         pattern  {DoFsub,,,Eprime}

; An "F" item consists of a "T" item optionally followed by "*" or "/"
; followed by another "T" item:

F               pattern  {sl_Match2, T,,Fprime}
Fprime          pattern  {MatchChar, '*', Fprime2, fmf}
fmf             pattern  {sl_Match2, T, 0, pMul}
pMul            pattern  {DoFmul,,,Fprime}
```

```
Fprime2         pattern  {MatchChar, '/', Succeed, fdf}
fdf             pattern  {sl_Match2, T, 0, pDiv}
pDiv            pattern  {DoFdiv, 0, 0,Fprime}

; T item consists of an "S" item or a "-" followed by another "T" item:

T               pattern  {MatchChar, '-', S, TT}
TT              pattern  {sl_Match2, T, 0,tpn}
tpn             pattern  {DoFchs}

; An "S" item is either a floating point constant or "(" followed by
; and "E" item followed by ")".
;
; The regular expression for a floating point constant is
;
;      [0-9]+ ( "." [0-9]* | ) ( ((e|E) (+|-| ) [0-9]+) | )
;
; Note: the pattern "Const" matches exactly the characters specified
;      by the above regular expression. It is the pattern the calc-
;      ulator grabs when converting a string to a floating point number.

Const           pattern  {sl_match2, ConstStr, 0, FLDConst}
ConstStr        pattern  {sl_match2, DoDigits, 0, Const2}
Const2          pattern  {matchchar, '.', Const4, Const3}
Const3          pattern  {sl_match2, DoDigits, Const4, Const4}
Const4          pattern  {matchchar, 'e', const5, const6}
Const5          pattern  {matchchar, 'E', Succeed, const6}
Const6          pattern  {matchchar, '+', const7, const8}
Const7          pattern  {matchchar, '-', const8, const8}
Const8          pattern  {sl_match2, DoDigits}

FldConst        pattern  {PushValue}

; DoDigits handles the regular expression [0-9]+

DoDigits        pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern  {Spancset, Digits}

; The S production handles constants or an expression in parentheses.

S               pattern  {MatchChar, '(', Const, IntE}
IntE            pattern  {sl_Match2, E, 0, CloseParen}
CloseParen      pattern  {MatchChar, ')'}

; The Succeed pattern always succeeds.

Succeed         pattern  {DoSucceed}


; We use digits from the UCR Standard Library cset standard sets.

                include  stdsets.a

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; DoSucceed matches the empty string. In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed       proc     far
                mov      ax, di
                stc
                ret
DoSucceed       endp
```

```
; DoFadd - Adds the two items on the top of the FPU stack.

DoFadd          proc    far
                faddp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc                             ;Always succeed.
                ret
DoFadd          endp

; DoFsub - Subtracts the two values on the top of the FPU stack.

DoFsub          proc    far
                fsubp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFsub          endp

; DoFmul- Multiplies the two values on the FPU stack.

DoFmul          proc    far
                fmulp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFmul          endp

; DoFdiv- Divides the two values on the FPU stack.

DoFDiv          proc    far
                fdivp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFDiv          endp

; DoFchs- Negates the value on the top of the FPU stack.

DoFchs          proc    far
                fchs
                mov     ax, di                  ;Required by sl_Match
                stc
                ret
DoFchs          endp


; PushValue-    We've just matched a string that corresponds to a
;               floating point constant. Convert it to a floating
;               point value and push that value onto the FPU stack.

PushValue       proc    far
                push    ds
                push    es
                pusha
                mov     ax, dseg
                mov     ds, ax

                lesi    Const           ;FP val matched by this pat.
                patgrab                 ;Get a copy of the string.
                atof                    ;Convert to real.
                free                    ;Return mem used by patgrab.
                lesi    CurValue        ;Copy floating point accumulator
                sdfpa                   ; to a local variable and then
                fld     CurValue        ; copy that value to the FPU stk.

                popa
                mov     ax, di
                pop     es
                pop     ds
```

```
                stc
                ret
PushValue       endp

; DoExp-        This routine expects a pointer to a string containing
;               an arithmetic expression in ES:DI. It evaluates the
;               given expression and prints the result.

DoExp           proc    near
                finit               ;Be sure to do this!
                fwait

                puts                ;Print the expression

                ldxi    Expression
                xor     cx, cx
                match
                jc      GoodVal
                printff
                byte    " is an illegal expression",cr,lf,0
                ret

GoodVal:        fstp    CurValue
                printff
                byte    " = %12.6ge\n",0
                dword   CurValue
                ret
DoExp           endp


; The main program tests the expression evaluator.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                lesi    Str1
                call    DoExp
                lesi    Str2
                call    DoExp
                lesi    Str3
                call    DoExp
                lesi    Str4
                call    DoExp
                lesi    Str5
                call    DoExp
                lesi    Str6
                call    DoExp
                lesi    Str7
                call    DoExp
                lesi    Str8
                call    DoExp
                lesi    Str9
                call    DoExp
                lesi    Str10
                call    DoExp

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
```

```
zzzzzzseg       ends
                end     Main
```

Sample Output:

```
5+2*(3-1) = 9.000E+0000
(5+2)*(7-10) = -2.100E+0001
5 = 5.000E+0000
(6+2)/(5+1)-7e5*2/1.3e2+1.5 = -1.077E+0004
2.5*(2-(3+1)/4+1) = 5.000E+0000
6+(-5*2) = -4.000E+0000
6*-1 = -6.000E+0000
1.2e5/2.1e5 = 5.714E-0001
0.9999999999999999+1e-15 = 1.000E+0000
2.1-1.1 = 1.000E+0000
```

16.8.4 A Tiny Assembler

Although the UCR Standard Library pattern matching routines would probably not be appropriate for writing a full lexical analyzer or compiler, they are useful for writing small compilers / assemblers or programs where speed of compilation / assembly is of little concern. One good example is the simple nonsymbolic assembler appearing in the SIM886[10] simulator for an earlier version of the x86 processors[11]. This "mini-assembler" accepts an x86 assembly language statement and immediately assembles it into memory. This allows SIM886 users to create simple assembly language programs within the SIM886 monitor / debugger[12]. Using the Standard Library pattern matching routines makes it very easy to implement such an assembler.

The grammar for this miniassembler is

```
Stmt →          Grp1 reg "," operand |
                Grp2 reg "," reg "," constant |
                Grp3 operand |
                goto operand |
                halt

Grp1 →          load | store | add | sub
Grp2 →          ifeq | iflt | ifgt
Grp3 →          get | put

reg →           ax | bx | cx | dx

operand →       reg | constant | [bx] | constant [bx]

constant →      hexdigit constant | hexdigit

hexdigit →      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b |
                c | d | e | f
```

There are some minor semantic details that the program handles (such as disallowing stores into immediate operands). The assembly code for the miniassembler follows:

```
; ASM.ASM
;

                .xlist
                include     stdlib.a
                matchfuncs
                includelib  stdlib.lib
                .list
```

10. SIM886 is an earlier version of SIMx86. It is also available on the Companion CD-ROM.
11. The current x86 system is written with Borland's Delphi, using a pattern matching library written for Pascal that is very similar to the Standard Library's pattern matching code.
12. See the lab manual for more details on SIM886.

```
dseg            segment  para public 'data'

; Some sample statements to assemble:

Str1            byte     "load ax, 0",0
Str2            byte     "load ax, bx",0
Str3            byte     "load ax, ax",0
Str4            byte     "add ax, 15",0
Str5            byte     "sub ax, [bx]",0
Str6            byte     "store bx, [1000]",0
Str7            byte     "load bx, 2000[bx]",0
Str8            byte     "goto 3000",0
Str9            byte     "iflt ax, bx, 100",0
Str10           byte     "halt",0
Str11           byte     "This is illegal",0
Str12           byte     "load ax, store",0
Str13           byte     "store ax, 1000",0
Str14           byte     "ifeq ax, 0, 0",0

; Variables used by the assembler.

AsmConst        word     0
AsmOpcode       byte     0
AsmOprnd1       byte     0
AsmOprnd2       byte     0

                include  stdsets.a    ;Bring in the standard char sets.

; Patterns for the assembler:

; Pattern is (
;       (load|store|add|sub) reg "," operand |
;       (ifeq|iflt|ifgt) reg1 "," reg2 "," const |
;       (get|put) operand |
;       goto operand |
;       halt
;       )
;
; With a few semantic additions (e.g., cannot store to a const).

InstrPat        pattern  {spancset, WhiteSpace,Grp1,Grp1}

Grp1            pattern  {sl_Match2,Grp1Strs, Grp2 ,Grp1Oprnds}
Grp1Strs        pattern  {TryLoad,,Grp1Store}
Grp1Store       pattern  {TryStore,,Grp1Add}
Grp1Add         pattern  {TryAdd,,Grp1Sub}
Grp1Sub         pattern  {TrySub}

; Patterns for the LOAD, STORE, ADD, and SUB instructions.

LoadPat         pattern  {MatchStr,LoadInstr2}
LoadInstr2      byte     "LOAD",0

StorePat        pattern  {MatchStr,StoreInstr2}
StoreInstr2     byte     "STORE",0

AddPat          pattern  {MatchStr,AddInstr2}
AddInstr2       byte     "ADD",0

SubPat          pattern  {MatchStr,SubInstr2}
SubInstr2       byte     "SUB",0

; Patterns for the group one (LOAD/STORE/ADD/SUB) instruction operands:

Grp1Oprnds      pattern  {spancset,WhiteSpace,Grp1reg,Grp1reg}
Grp1Reg         pattern  {MatchReg,AsmOprnd1,,Grp1ws2}
Grp1ws2         pattern  {spancset,WhiteSpace,Grp1Comma,Grp1Comma}
Grp1Comma       pattern  {MatchChar,',',0,Grp1ws3}
Grp1ws3         pattern  {spancset,WhiteSpace,Grp1Op2,Grp1Op2}
```

```
Grp1Op2          pattern  {MatchGen,,,EndOfLine}
EndOfLine        pattern  {spancset,WhiteSpace,NullChar,NullChar}
NullChar         pattern  {EOS}

Grp1Op2Reg       pattern  {MatchReg,AsmOprnd2}

; Patterns for the group two instructions (IFEQ, IFLT, IFGT):

Grp2             pattern  {sl_Match2,Grp2Strs, Grp3 ,Grp2Oprnds}
Grp2Strs         pattern  {TryIFEQ,,Grp2IFLT}
Grp2IFLT         pattern  {TryIFLT,,Grp2IFGT}
Grp2IFGT         pattern  {TryIFGT}

Grp2Oprnds       pattern  {spancset,WhiteSpace,Grp2reg,Grp2reg}
Grp2Reg          pattern  {MatchReg,AsmOprnd1,,Grp2ws2}
Grp2ws2          pattern  {spancset,WhiteSpace,Grp2Comma,Grp2Comma}
Grp2Comma        pattern  {MatchChar,',',0,Grp2ws3}
Grp2ws3          pattern  {spancset,WhiteSpace,Grp2Reg2,Grp2Reg2}
Grp2Reg2         pattern  {MatchReg,AsmOprnd2,,Grp2ws4}
Grp2ws4          pattern  {spancset,WhiteSpace,Grp2Comma2,Grp2Comma2}
Grp2Comma2       pattern  {MatchChar,',',0,Grp2ws5}
Grp2ws5          pattern  {spancset,WhiteSpace,Grp2Op3,Grp2Op3}
Grp2Op3          pattern  {ConstPat,,,EndOfLine}

; Patterns for the IFEQ, IFLT, and IFGT instructions.

IFEQPat          pattern  {MatchStr,IFEQInstr2}
IFEQInstr2       byte     "IFEQ",0

IFLTPat          pattern  {MatchStr,IFLTInstr2}
IFLTInstr2       byte     "IFLT",0

IFGTPat          pattern  {MatchStr,IFGTInstr2}
IFGTInstr2       byte     "IFGT",0

; Grp3 Patterns:

Grp3             pattern  {sl_Match2,Grp3Strs, Grp4 ,Grp3Oprnds}
Grp3Strs         pattern  {TryGet,,Grp3Put}
Grp3Put          pattern  {TryPut,,Grp3GOTO}
Grp3Goto         pattern  {TryGOTO}

; Patterns for the GET and PUT instructions.

GetPat           pattern  {MatchStr,GetInstr2}
GetInstr2        byte     "GET",0

PutPat           pattern  {MatchStr,PutInstr2}
PutInstr2        byte     "PUT",0

GOTOPat          pattern  {MatchStr,GOTOInstr2}
GOTOInstr2       byte     "GOTO",0

; Patterns for the group three (PUT/GET/GOTO) instruction operands:

Grp3Oprnds       pattern  {spancset,WhiteSpace,Grp3Op,Grp3Op}
Grp3Op           pattern  {MatchGen,,,EndOfLine}

; Patterns for the group four instruction (HALT).

Grp4             pattern  {TryHalt,,,EndOfLine}

HaltPat          pattern  {MatchStr,HaltInstr2}
HaltInstr2       byte     "HALT",0

; Patterns to match the four non-register addressing modes:

BXIndrctPat      pattern  {MatchStr,BXIndrctStr}
BXIndrctStr      byte     "[BX]",0
```

```
BXIndexedPat    pattern  {ConstPat,,,BXIndrctPat}

DirectPat       pattern  {MatchChar,'[',,DP2}
DP2             pattern  {ConstPat,,,DP3}
DP3             pattern  {MatchChar,']'}

ImmediatePat    pattern  {ConstPat}

; Pattern to match a hex constant:

HexConstPat     pattern  {Spancset, xdigits}

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; The store macro tweaks the DS register and stores into the
; specified variable in DSEG.

store           macro    Where, What
                push     ds
                push     ax
                mov      ax, seg Where
                mov      ds, ax
                mov      Where, What
                pop      ax
                pop      ds
                endm

; Pattern matching routines for the assembler.
; Each mnemonic has its own corresponding matching function that
; attempts to match the mnemonic. If it does, it initializes the
; AsmOpcode variable with the base opcode of the instruction.

; Compare against the "LOAD" string.

TryLoad         proc     far
                push     dx
                push     si
                ldxi     LoadPat
                match2
                jnc      NoTLMatch

                store    AsmOpcode, 0           ;Initialize base opcode.

NoTLMatch:      pop      si
                pop      dx
                ret
TryLoad         endp

; Compare against the "STORE" string.

TryStore        proc     far
                push     dx
                push     si
                ldxi     StorePat
                match2
                jnc      NoTSMatch
                store    AsmOpcode, 1           ;Initialize base opcode.

NoTSMatch:      pop      si
                pop      dx
                ret
TryStore        endp

; Compare against the "ADD" string.

TryAdd          proc     far
                push     dx
```

```
                push    si
                ldxi    AddPat
                match2
                jnc     NoTAMatch
                store   AsmOpcode, 2            ;Initialize ADD opcode.

NoTAMatch:      pop     si
                pop     dx
                ret
TryAdd          endp

; Compare against the “SUB” string.

TrySub          proc    far
                push    dx
                push    si
                ldxi    SubPat
                match2
                jnc     NoTMMatch
                store   AsmOpcode, 3            ;Initialize SUB opcode.

NoTMMatch:      pop     si
                pop     dx
                ret
TrySub          endp

; Compare against the “IFEQ” string.

TryIFEQ         proc    far
                push    dx
                push    si
                ldxi    IFEQPat
                match2
                jnc     NoIEMatch
                store   AsmOpcode, 4            ;Initialize IFEQ opcode.

NoIEMatch:      pop     si
                pop     dx
                ret
TryIFEQ         endp

; Compare against the “IFLT” string.

TryIFLT         proc    far
                push    dx
                push    si
                ldxi    IFLTPat
                match2
                jnc     NoILMatch
                store   AsmOpcode, 5            ;Initialize IFLT opcode.

NoILMatch:      pop     si
                pop     dx
                ret
TryIFLT         endp

; Compare against the “IFGT” string.

TryIFGT         proc    far
                push    dx
                push    si
                ldxi    IFGTPat
                match2
                jnc     NoIGMatch
                store   AsmOpcode, 6            ;Initialize IFGT opcode.

NoIGMatch:      pop     si
                pop     dx
                ret
TryIFGT         endp
```

```
; Compare against the “GET” string.

TryGET          proc    far
                push    dx
                push    si
                ldxi    GetPat
                match2
                jnc     NoGMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 2            ;GET’s Special opcode.

NoGMatch:       pop     si
                pop     dx
                ret
TryGET          endp

; Compare against the “PUT” string.

TryPut          proc    far
                push    dx
                push    si
                ldxi    PutPat
                match2
                jnc     NoPMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 3            ;PUT’s Special opcode.

NoPMatch:       pop     si
                pop     dx
                ret
TryPUT          endp

; Compare against the “GOTO” string.

TryGOTO         proc    far
                push    dx
                push    si
                ldxi    GOTOPat
                match2
                jnc     NoGMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 1            ;PUT’s Special opcode.

NoGMatch:       pop     si
                pop     dx
                ret
TryGOTO         endp

; Compare against the “HALT” string.

TryHalt         proc    far
                push    dx
                push    si
                ldxi    HaltPat
                match2
                jnc     NoHMatch
                store   AsmOpcode, 7            ;Initialize Special opcode.
                store   AsmOprnd1, 0            ;Halt’s special opcode.
                store   AsmOprnd2, 0

NoHMatch:       pop     si
                pop     dx
                ret
TryHALT         endp

; MatchReg checks to see if we’ve got a valid register value. On entry,
; DS:SI points at the location to store the byte opcode (0, 1, 2, or 3) for
; a reasonable register (AX, BX, CX, or DX); ES:DI points at the string
; containing (hopefully) the register operand, and CX points at the last
```

```
; location plus one we can check in the string.
;
; On return, Carry=1 for success, 0 for failure. ES:AX must point beyond
; the characters which make up the register if we have a match.

MatchReg        proc    far

; ES:DI Points at two characters which should be AX/BX/CX/DX. Anything
; else is an error.

                cmp     byte ptr es:1[di], 'X'          ;Everyone needs this
                jne     BadReg
                xor     ax, ax                          ;886 "AX" reg code.
                cmp     byte ptr es:[di], 'A'           ;AX?
                je      GoodReg
                inc     ax
                cmp     byte ptr es:[di], 'B'           ;BX?
                je      GoodReg
                inc     ax
                cmp     byte ptr es:[di], 'C'           ;CX?
                je      GoodReg
                inc     ax
                cmp     byte ptr es:[di], 'D'           ;DX?
                je      GoodReg
BadReg:         clc
                mov     ax, di
                ret

GoodReg:
                mov     ds:[si], al             ;Save register opcode.
                lea     ax, 2[di]               ;Skip past register.
                cmp     ax, cx                  ;Be sure we didn't go
                ja      BadReg                  ; too far.
                stc
                ret
MatchReg        endp

; MatchGen-     Matches a general addressing mode. Stuffs the appropriate
;               addressing mode code into AsmOprnd2. If a 16-bit constant
;               is required by this addressing mode, this code shoves that
;               into the AsmConst variable.

MatchGen        proc    far
                push    dx
                push    si

; Try a register operand.

                ldxi    Grp1Op2Reg
                match2
                jc      MGDone

; Try "[bx]".

                ldxi    BXIndrctPat
                match2
                jnc     TryBXIndexed
                store   AsmOprnd2, 4
                jmp     MGDone

; Look for an operand of the form "xxxx[bx]".

TryBXIndexed:
                ldxi    BXIndexedPat
                match2
                jnc     TryDirect
                store   AsmOprnd2, 5
                jmp     MGDone

; Try a direct address operand "[xxxx]".
```

```
TryDirect:
                ldxi    DirectPat
                match2
                jnc     TryImmediate
                store   AsmOprnd2, 6
                jmp     MGDone

; Look for an immediate operand “xxxx”.

TryImmediate:
                ldxi    ImmediatePat
                match2
                jnc     MGDone
                store   AsmOprnd2, 7

MGDone:
                pop     si
                pop     dx
                ret
MatchGen        endp

; ConstPat-     Matches a 16-bit hex constant. If it matches, it converts
;               the string to an integer and stores it into AsmConst.

ConstPat        proc    far
                push    dx
                push    si
                ldxi    HexConstPat
                match2
                jnc     CPDone

                push    ds
                push    ax
                mov     ax, seg AsmConst
                mov     ds, ax
                atoh
                mov     AsmConst, ax
                pop     ax
                pop     ds
                stc

CPDone:         pop     si
                pop     dx
                ret
ConstPat        endp

; Assemble-     This code assembles the instruction that ES:DI points
;               at and displays the hex opcode(s) for that instruction.

Assemble        proc    near

; Print out the instruction we’re about to assemble.

                print
                byte    “Assembling: “,0
                strupr
                puts
                putcr

; Assemble the instruction:

                ldxi    InstrPat
                xor     cx, cx
                match
                jnc     SyntaxError

; Quick check for illegal instructions:

                cmp     AsmOpcode, 7                    ;Special/Get instr.
```

```
                jne     TryStoreInstr
                cmp     AsmOprnd1, 2                    ;GET opcode
                je      SeeIfImm
                cmp     AsmOprnd1, 1                    ;Goto opcode
                je      IsGOTO

TryStoreInstr:  cmp     AsmOpcode, 1                    ;Store Instruction
                jne     InstrOkay

SeeIfImm:       cmp     AsmOprnd2, 7                    ;Immediate Adrs Mode
                jne     InstrOkay
                print
                db      "Syntax error: store/get immediate not allowed."
                db      " Try Again",cr,lf,0
                jmp     ASMDone

IsGOTO:         cmp     AsmOprnd2, 7            ;Immediate mode for GOTO
                je      InstrOkay
                print
                db      "Syntax error: GOTO only allows immediate "
                byte    "mode.",cr,lf
                db      0
                jmp     ASMDone

; Merge the opcode and operand fields together in the instruction byte,
; then output the opcode byte.

InstrOkay:      mov     al, AsmOpcode
                shl     al, 1
                shl     al, 1
                or      al, AsmOprnd1
                shl     al, 1
                shl     al, 1
                shl     al, 1
                or      al, AsmOprnd2
                puth
                cmp     AsmOpcode, 4                    ;IFEQ instruction
                jb      SimpleInstr
                cmp     AsmOpcode, 6                    ;IFGT instruction
                jbe     PutConstant

SimpleInstr:    cmp     AsmOprnd2, 5
                jb ASMDone

; If this instruction has a 16 bit operand, output it here.

PutConstant:    mov     al, ' '
                putc
                mov     ax, ASMConst
                puth
                mov     al, ' '
                putc
                xchg    al, ah
                puth
                jmp     ASMDone

SyntaxError:    print
                db      "Syntax error in instruction."
                db      cr,lf,0

ASMDone:        putcr
                ret
Assemble        endp

; Main program that tests the assembler.

Main            proc
                mov     ax, seg dseg ;Set up the segment registers
                mov     ds, ax
                mov     es, ax
```

```
                        meminit

                        lesi     Str1
                        call     Assemble
                        lesi     Str2
                        call     Assemble
                        lesi     Str3
                        call     Assemble
                        lesi     Str4
                        call     Assemble
                        lesi     Str5
                        call     Assemble
                        lesi     Str6
                        call     Assemble
                        lesi     Str7
                        call     Assemble
                        lesi     Str8
                        call     Assemble
                        lesi     Str9
                        call     Assemble
                        lesi     Str10
                        call     Assemble
                        lesi     Str11
                        call     Assemble
                        lesi     Str12
                        call     Assemble
                        lesi     Str13
                        call     Assemble
                        lesi     Str14
                        call     Assemble

Quit:                   ExitPgm
Main                    endp
cseg                    ends

sseg                    segment  para stack 'stack'
stk                     db       256 dup ("stack ")
sseg                    ends

zzzzzzseg               segment  para public 'zzzzzz'
LastBytes               db       16 dup (?)
zzzzzzseg               ends
                        end      Main
```

Sample Output:

```
Assembling: LOAD AX, 0
07 00 00
Assembling: LOAD AX, BX
01
Assembling: LOAD AX, AX
00
Assembling: ADD AX, 15
47 15 00
Assembling: SUB AX, [BX]
64
Assembling: STORE BX, [1000]
2E 00 10
Assembling: LOAD BX, 2000[BX]
0D 00 20
Assembling: GOTO 3000
EF 00 30
Assembling: IFLT AX, BX, 100
A1 00 01
Assembling: HALT
E0
Assembling: THIS IS ILLEGAL
Syntax error in instruction.
```

```
Assembling: LOAD AX, STORE
Syntax error in instruction.

Assembling: STORE AX, 1000
Syntax error: store/get immediate not allowed. Try Again

Assembling: IFEQ AX, 0, 0
Syntax error in instruction.
```

16.8.5 The "MADVENTURE" Game

Computer games are a perfect example of programs that often use pattern matching. One class of computer games in general, the *adventure* game[13], is a perfect example of games that use pattern matching. An adventure style game excepts English-like commands from the user, parses these commands, and acts upon them. In this section we will develop an adventure game *shell*. That is, it will be a reasonably functional adventure style game, capable of accepting and processing user commands. All you need do is supply a story line and a few additional details to develop a fully functioning adventure class game.

An adventure game usually consists of some sort of *maze* through which the player moves. The program processes commands like *go north* or *go right* to move the player through the maze. Each move can deposit the player in a new room of the game. Generally, each room or area contains objects the player can interact with. This could be reward objects such as items of value or it could be an antagonistic object like a monster or enemy player.

Usually, an adventure game is a *puzzle* of some sort. The player finds clues and picks up useful object in one part of the maze to solve problems in other parts of the maze. For example, a player could pick up a key in one room that opens a chest in another; then the player could find an object in the chest that is useful elsewhere in the maze. The purpose of the game is to solve all the interlocking puzzles and maximize one's score (however that is done). This text will not dwell upon the subtleties of game design; that is a subject for a different text. Instead, we'll look at the tools and data structures required to implement the game design.

The Madventure game's use of pattern matching is quite different from the previous examples appearing in this chapter. In the examples up to this point, the matching routines specifically checked the validity of an input string; Madventure does not do this. Instead, it uses the pattern matching routines to simply determine if certain key words appear on a line input by the user. The program handles the actual parsing (determining if the command is syntactically correct). To understand how the Madventure game does this, it would help if we took a look at how to play the Madventure game[14].

The Madventure prompts the user to enter a command. Unlike the original adventure game that required commands like "GO NORTH" (with no other characters other than spaces as part of the command), Madventure allows you to write whole sentences and then it attempts to pick out the key words from those sentences. For example, Madventure accepts the "GO NORTH" command; however, it also accepts commands like "North is the direction I want to go" and "I want to go in the north direction." Madventure doesn't really care as long as it can find "GO" and "NORTH" *somewhere* on the command line. This is a little more flexible that the original Adventure game structure. Of course, this scheme isn't infallible, it will treat commands like "I absolutely, positively, do *NOT* want to go anywhere near the north direction" as a "GO NORTH" command. Oh well, the user almost always types just "GO NORTH" anyway.

13. These are called adventure games because the original program of the genre was called "Adventure."
14. One word of caution, no one is going to claim that Madventure is a great game. If it were, it would be sold, it wouldn't appear in this text! So don't expect too much from the design of the game itself.

A Madventure command usually consists of a *noun* keyword and a *verb* keyword. The Madventure recognizes six verbs and fourteen nouns[15]. The verbs are

```
verbs →          go | get | drop | inventory | quit | help
```

The nouns are

```
nouns →          north | south | east | west | lime | beer | card |
                 sign | program | homework | money | form | coupon
```

Obviously, Madventure does not allow all combinations of verbs and nouns. Indeed, the following patterns are the only legal ones:

```
LegalCmds →      go direction | get item | drop item | inventory |
                 quit | help

direction →      north | south | east | west

item →           lime | beer | card | sign | program | homework |
                 money | form | coupon
```

However, the pattern does not enforce this grammar. It just locates a noun and a verb on the line and, if found, sets the noun and verb variables to appropriate values to denote the keywords it finds. By letting the main program handle the parsing, the program is somewhat more flexible.

There are two main patterns in the Madventure program: NounPat and VerbPat. These patterns match words (nouns or verbs) using a regular expression like the following:

```
(ARB* ' ' | ε) word (' ' | EOS)
```

This regular expression matches a word that appears at the beginning of a sentence, at the end of a sentence, anywhere in the middle of a sentence, or a sentence consisting of a single word. Madventure uses a macro (MatchNoun or MatchVerb) to create an expression for each noun and verb in the above expression.

To get an idea of how Madvent processes words, consider the following VerbPat pattern:

```
VerbPat        pattern    {sl_match2, MatchGo}
               MatchVerb  MatchGO, MatchGet, "GO", 1
               MatchVerb  MatchGet, MatchDrop, "GET", 2
               MatchVerb  MatchDrop, MatchInv, "DROP", 3
               MatchVerb  MatchInv, MatchQuit, "INVENTORY", 4
               MatchVerb  MatchQuit, MatchHelp, "QUIT", 5
               MatchVerb  MatchHelp, 0, "HELP", 6
```

The MatchVerb macro expects four parameters. The first is an arbitrary pattern name; the second is a link to the next pattern in the list; the third is the string to match, and the fourth is a number that the matching routines will store into the verb variable if that string matches (by default, the verb variable contains zero). It is very easy to add new verbs to this list. For example, if you wanted to allow "run" and "walk" as synonyms for the "go" verb, you would just add two patterns to this list:

```
VerbPat        pattern       {sl_match2, MatchGo}
               MatchVerb  MatchGO, MatchGet, "GO", 1
               MatchVerb  MatchGet, MatchDrop, "GET", 2
               MatchVerb  MatchDrop, MatchInv, "DROP", 3
               MatchVerb  MatchInv, MatchQuit, "INVENTORY", 4
               MatchVerb  MatchQuit, MatchHelp, "QUIT", 5
               MatchVerb  MatchHelp, MatchRun, "HELP", 6
               MatchVerb  MatchRun, MatchWalk, "RUN", 1
               MatchVerb  MatchWalk, 0, "WALK", 1
```

There are only two things to consider when adding new verbs: first, don't forget that the next field of the last verb should contain zero; second, the current version of Madventure

15. However, one beautiful thing about Madventure is that it is *very* easy to extend and add more nouns and verbs.

only allows up to seven verbs. If you want to add more you will need to make a slight modification to the main program (more on that, later). Of course, if you only want to create synonyms, as we've done here, you simply reuse existing verb values so there is no need to modify the main program.

When you call the match routine and pass it the address of the VerbPat pattern, it scans through the input string looking for the first verb. If it finds that verb ("GO") it sets the verb variable to the corresponding verb value at the end of the pattern. If match cannot find the first verb, it tries the second. If that fails, it tries the third, and so on. If match cannot find *any* of the verbs in the input string, it does not modify the verb variable (which contains zero). If there are *two* or more of the above verbs on the input line, match will locate the first verb in the verb list above. *This may not be the first verb appearing on the line.* For example, if you say "Let's get the money and go north" the match routine will match the "go" verb, not the "get" verb. By the same token, the NounPat pattern would match the north noun, not the money noun. So this command would be identical to "GO NORTH."

The MatchNoun is almost identical to the MatchVerb macro; there is, however, one difference – the MatchNoun macro has an extra parameter which is the name of the data structure representing the given object (if there is one). Basically, all the nouns (in this version of Madventure) except NORTH, SOUTH, EAST, and WEST have some sort of data structure associated with them.

The maze in Madventure consists of nine rooms defined by the data structure:

```
Room            struct
north           word    ?
south           word    ?
west            word    ?
east            word    ?
ItemList        word    MaxWeight dup (?)
Description     word    ?
Room            ends
```

The north, south, west, and east fields contain near pointers to other rooms. The program uses the CurRoom variable to keep track of the player's current position in the maze. When the player issues a "GO" command of some sort, Madventure copies the appropriate value from the north, south, west, or east field to the CurRoom variable, effectively changing the room the user is in. If one of these pointers is NULL, then the user cannot move in that direction.

The direction pointers are independent of one another. If you issue the command "GO NORTH" and then issue the command "GO SOUTH" upon arriving in the new room, there is no guarantee that you will wind up in the original room. The south field of the second room may not point at the room that led you there. Indeed, there are several cases in the Madventure game where this occurs.

The ItemList array contains a list of near pointers to objects that could be in the room. In the current version of this game, the objects are all the nouns except *north*, *south*, *east*, and *west*. The player can carry these objects from room to room (indeed, that is the major purpose of this game). Up to MaxWeight objects can appear in the room (MaxWeight is an assembly time constant that is currently four; so there are a maximum of four items in any one given room). If an entry in the ItemList is non-NULL, then it is a pointer to an Item object. There may be zero to MaxWeight objects in a room.

The Description field contains a pointer to a zero terminated string that describes the room. The program prints this string each time through the command loop to keep the player oriented.

The second major data type in Madventure is the Item structure. This structure takes the form:

```
Item            struct
Value           word    ?
Weight          word    ?
Key             word    ?
ShortDesc       word    ?
LongDesc        word    ?
WinDesc         word    ?
Item            ends
```

The Value field contains an integer value awarded to the player when the player drops this object in the appropriate room. This is how the user scores points.

The Weight field usually contains one or two and determines how much this object "weighs." The user can only carry around MaxWeight units of weight at any one given time. Each time the user picks up an object, the weight of that object is added to the user's total weight. When the user drops an object, Madventure subtracts the object's weight from the total.

The Key field contains a pointer to a room associated with the object. When the user drops the object in the Key room, the user is awarded the points in the Value field and the object disappears from the game. If the user drops the object in some other room, the object stays in that room until the user picks it up again.

The ShortDesc, LongDesc, and WinDesc fields contain pointers to zero terminated strings. Madventure prints the ShortDesc string in response to an INVENTORY command. It prints the LongDesc string when describing a room's contents. It prints the WinDesc string when the user drops the object in its Key room and the object disappears from the game.

The Madventure main program is deceptively simple. Most of the logic is hidden in the pattern matching routines and in the parsing routine. We've already discussed the pattern matching code; the only important thing to remember is that it initializes the noun and verb variables with a value uniquely identifying each noun and verb. The main program's logic uses these two values as an index into a two dimensional table that takes the following form:

Table 65: Madventure Noun/Verb Table

	No Verb	GO	GET	DROP	Inven-tory	Quit	Help
No Noun					Inven-tory	Quit	Help
North		Do North					
South		Do South					
East		Do East					
West		Do West					
Lime			Get Item	Drop Item			
Beer			Get Item	Drop Item			
Card			Get Item	Drop Item			
Sign			Get Item	Drop Item			
Program			Get Item	Drop Item			

Table 65: Madventure Noun/Verb Table

	No Verb	GO	GET	DROP	Inven-tory	Quit	Help
Home-work			Get Item	Drop Item			
Money			Get Item	Drop Item			
Form			Get Item	Drop Item			
Coupon			Get Item	Drop Item			

The empty entries in this table correspond to illegal commands. The other entries are addresses of code within the main program that handles the given command.

To add more nouns (objects) to the game, you need only extend the NounPat pattern and add additional rows to the table (of course, you may need to add code to handle the new objects if they are not easily handled by the routines above). To add new verbs you need only extended the VerbPat pattern and add new columns to this table[16].

Other than the goodies mentioned above, the rest of the program utilizes techniques appearing throughout this and previous chapters. The only real surprising thing about this program is that you can implement a fairly complex program with so few lines of code. But such is the advantage of using pattern matching techniques in your assembly language programs.

```
; MADVENT.ASM
;
; This is a “shell” of an adventure game that you can use to create
; your own adventure style games.

                .xlist
                .286
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list

dseg            segment     para public ‘data’

; Equates:

NULL            equ      0
MaxWeight       equ      4           ;Max weight user can carry at one time.


; The “ROOM” data structure defines a room, or area, where a player can
; go. The NORTH, SOUTH, EAST, and WEST fields contain the address of
; the rooms to the north, south, east, and west of the room. The game
; transfers control to the room whose address appears in these fields
; when the player supplies a GO NORTH, GO SOUTH, etc., command.
;
; The ITEMLIST field contains a list of pointers to objects appearing
; in this room. In this game, the user can pick up and drop these
; objects (if there are any present).
;
; The DESCRIPTION field contains a (near) address of a short description
; of the current room/area.
```

16. Currently, the Madventure program computes the index into this table (a 14x8) table by shifting to the left three bits rather than multiplying by eight. You will need to modify this code if you add more columns to the table.

```
Room            struct
north           word    ? ;Near pointers to other structures where
south           word    ? ; we will wind up on the GO NORTH, GO SOUTH,
west            word    ? ; etc., commands.
east            word    ?

ItemList        word    MaxWeight dup (?)

Description     word    ? ;Description of room.
Room            ends


; The ITEM data structure describes the objects that may appear
; within a room (in the ITEMLIST above). The VALUE field contains
; the number of points this object is worth if the user drops it
; off in the proper room (i.e, solves the puzzle). The WEIGHT
; field provides the weight of this object. The user can only
; carry four units of weight at a time. This field is usually
; one, but may be more for larger objects. The KEY field is the
; address of the room where this object must be dropped to solve
; the problem. The SHORTDESC field is a pointer to a string that
; the program prints when the user executes an INVENTORY command.
; LONGDESC is a pointer to a string the program prints when des-
; cribing the contents of a room. The WINDESC field is a pointer
; to a string that the program prints when the user solves the
; appropriate puzzle.

Item            struct
Value           word    ?
Weight          word    ?
Key             word    ?
ShortDesc       word    ?
LongDesc        word    ?
WinDesc         word    ?
Item            ends


; State variables for the player:

CurRoom         word      Room1                 ;Room the player is in.
ItemsOnHand     word      MaxWeight dup (?)     ;Items the player carries.
CurWeight       word      0                     ;Weight of items carried.
CurScore        word      15                    ;Player's current score.
TotalCounter    word      9                     ;Items left to place.
Noun            word      0                     ;Current noun value.
Verb            word      0                     ;Current verb value.
NounPtr         word      0                     ;Ptr to current noun item.


; Input buffer for commands

InputLine       byte    128 dup (?)
; The following macros generate a pattern which will match a single word
; which appears anywhere on a line. In particular, they match a word
; at the beginning of a line, somewhere in the middle of the line, or
; at the end of a line. This program defines a word as any sequence
; of character surrounded by spaces or the beginning or end of a line.
;
; MatchNoun/Verb matches lines defined by the regular expression:
;
;      (ARB* ' ' | ε) string (' ' | EOS)

MatchNoun       macro     Name, next, WordString, ItemVal, ItemPtr
                local     WS1, WS2, WS3, WS4
                local     WS5, WS6, WordStr

Name            Pattern   {sl_match2, WS1, next}
WS1             Pattern   {MatchStr, WordStr, WS2, WS5}
WS2             Pattern   {arb,0,0,WS3}
WS3             Pattern   {Matchchar, ' ',0, WS4}
```

```
WS4             Pattern     {MatchStr, WordStr, 0, WS5}
WS5             Pattern     {SetNoun,ItemVal,0,WS6}
WS6             Pattern     {SetPtr, ItemPtr,0,MatchEOS}
WordStr         byte        WordString
                byte        0
                endm


MatchVerb       macro       Name, next, WordString, ItemVal
                local       WS1, WS2, WS3, WS4
                local       WS5, WordStr

Name            Pattern     {sl_match2, WS1, next}
WS1             Pattern     {MatchStr, WordStr, WS2, WS5}
WS2             Pattern     {arb,0,0,WS3}
WS3             Pattern     {Matchchar, ' ',0, WS4}
WS4             Pattern     {MatchStr, WordStr, 0, WS5}
WS5             Pattern     {SetVerb,ItemVal,0,MatchEOS}
WordStr         byte        WordString
                byte        0
                endm



; Generic patterns which most of the patterns use:

MatchEOS        Pattern     {EOS,0,MatchSpc}
MatchSpc        Pattern     {MatchChar,' '}


; Here are the list of nouns allowed in this program.

NounPat         pattern     {sl_match2, MatchNorth}

                MatchNoun   MatchNorth, MatchSouth, "NORTH", 1, 0
                MatchNoun   MatchSouth, MatchEast, "SOUTH", 2, 0
                MatchNoun   MatchEast, MatchWest, "EAST", 3, 0
                MatchNoun   MatchWest, MatchLime, "WEST", 4, 0
                MatchNoun   MatchLime, MatchBeer, "LIME", 5, Item3
                MatchNoun   MatchBeer, MatchCard, "BEER", 6, Item9
                MatchNoun   MatchCard, MatchSign, "CARD", 7, Item2
                MatchNoun   MatchSign, MatchPgm, "SIGN", 8, Item1
                MatchNoun   MatchPgm, MatchHW, "PROGRAM", 9, Item7
                MatchNoun   MatchHW, MatchMoney, "HOMEWORK", 10, Item4
                MatchNoun   MatchMoney, MatchForm, "MONEY", 11, Item5
                MatchNoun   MatchForm, MatchCoupon, "FORM", 12, Item6
                MatchNoun   MatchCoupon, 0, "COUPON", 13, Item8


; Here is the list of allowable verbs.

VerbPat         pattern     {sl_match2, MatchGo}

                MatchVerb   MatchGO, MatchGet, "GO", 1
                MatchVerb   MatchGet, MatchDrop, "GET", 2
                MatchVerb   MatchDrop, MatchInv, "DROP", 3
                MatchVerb   MatchInv, MatchQuit, "INVENTORY", 4
                MatchVerb   MatchQuit, MatchHelp, "QUIT", 5
                MatchVerb   MatchHelp, 0, "HELP", 6

; Data structures for the "maze".

Room1           room        {Room1, Room5, Room4, Room2,
                             {Item1,0,0,0},
                             Room1Desc}

Room1Desc       byte        "at the Commons",0

Item1           item        {10,2,Room3,GS1,GS2,GS3}
```

```
GS1             byte        “a big sign”,0
GS2             byte        “a big sign made of styrofoam with funny “
                byte        “letters on it.”,0
GS3             byte        “The ETA PI Fraternity thanks you for return”
                byte        “ing their sign, they”,cr,lf
                byte        “make you an honorary life member, as long as “
                byte        “you continue to pay”,cr,lf
                byte        “your $30 monthly dues, that is.”,0

Room2           room        {NULL, Room5, Room1, Room3,
                             {Item2,0,0,0},
                             Room2Desc}

Room2Desc       byte        ‘at the “C” on the hill above campus’,0

Item2           item        {10,1,Room1,LC1,LC2,LC3}
LC1             byte        “a lunch card”,0
LC2             byte        “a lunch card which someone must have “
                byte        “accidentally dropped here.”, 0
LC3             byte        “You get a big meal at the Commons cafeteria”
                byte        cr,lf
                byte        “It would be a good idea to go visit the “
                byte        “student health center”,cr,lf
                byte        “at this time.”,0

Room3           room        {NULL, Room6, Room2, Room2,
                             {Item3,0,0,0},
                             Room3Desc}

Room3Desc       byte        “at ETA PI Frat House”,0

Item3           item        {10,2,Room2,BL1,BL2,BL3}
BL1             byte        “a bag of lime”,0
BL2             byte        “a bag of baseball field lime which someone “
                byte        “is obviously saving for”,cr,lf
                byte        “a special occasion.”,0
BL3             byte        “You spread the lime out forming a big ‘++’ “
                byte        “after the ‘C’”,cr,lf
                byte        “Your friends in Computer Science hold you “
                byte        “in total awe.”,0

Room4           room        {Room1, Room7, Room7, Room5,
                             {Item4,0,0,0},
                             Room4Desc}

Room4Desc       byte        “in Dr. John Smith’s Office”,0

Item4           item        {10,1,Room7,HW1,HW2,HW3}
HW1             byte        “a homework assignment”,0
HW2             byte        “a homework assignment which appears to “
                byte        “to contain assembly language”,0
HW3             byte        “The grader notes that your homework “
                byte        “assignment looks quite”,cr,lf
                byte        “similar to someone else’s assignment “
                byte        “in the class and reports you”,cr,lf
                byte        “to the instructor.”,0

Room5           room        {Room1, Room9, Room7, Room2,
                             {Item5,0,0,0},
                             Room5Desc}

Room5Desc       byte         “in the computer lab”,0

Item5           item        {10,1,Room9,M1,M2,M3}
M1              byte        “some money”,0
M2              byte        “several dollars in an envelope in the “
                byte        “trashcan”,0
M3              byte        “The waitress thanks you for your “
                byte        “generous tip and gets you”,cr,lf
                byte        “another pitcher of beer. “
```

```
                byte        "Then she asks for your ID.",cr,lf
                byte        "You are at least 21 aren't you?",0

Room6           room        {Room3, Room9, Room5, NULL,
                             {Item6,0,0,0},
                             Room6Desc}

Room6Desc       byte        "at the campus book store",0

Item6           item        {10,1,Room8,AD1,AD2,AD3}
AD1             byte        "an add/drop/change form",0
AD2             byte        "an add/drop/change form filled out for "
                byte        "assembly to get a letter grade",0
AD3             byte        "You got the form in just in time. "
                byte        "It would have been a shame to",cr,lf
                byte        "have had to retake assembly because "
                byte        "you didn't realize you needed to ",cr,lf
                byte        "get a letter grade in the course.",0

Room7           room        {Room1, Room7, Room4, Room8,
                             {Item7,0,0,0},
                             Room7Desc}

Room7Desc       byte         "in the assembly lecture",0

Item7           item        {10,1,Room5,AP1,AP2,AP3}
AP1             byte        "an assembly language program",0
AP2             byte        "an assembly language program due in "
                byte        "the assemblylanguage class.",0
AP3             byte        "The sample program the instructor gave "
                byte        "you provided all the information",cr,lf
                byte        "you needed to complete your assignment. "
                byte        "You finish your work and",cr,lf
                byte        "head to the local pub to celebrate."
                byte        cr,lf,0

Room8           room        {Room5, Room6, Room7, Room9,
                             {Item8,0,0,0},
                             Room8Desc}

Room8Desc       byte         "at the Registrar's office",0

Item8           item        {10,1,Room6,C1,C2,C3}
C1              byte        "a coupon",0
C2              byte        "a coupon good for a free text book",0
C3              byte        'You get a free copy of "Cliff Notes for '
                byte        'The Art of Assembly',cr,lf
                byte        'Language Programming" Alas, it does not '
                byte        "provide all the",cr,lf
                byte        "information you need for the class, so you "
                byte        "sell it back during",cr,lf
                byte        "the book buy-back period.",0



Room9           room        {Room6, Room9, Room8, Room3,
                             {Item9,0,0,0},
                             Room9Desc}

Room9Desc       byte        "at The Pub",0
Item9           item        {10,2,Room4,B1,B2,B3}
B1              byte        "a pitcher of beer",0
B2              byte        "an ice cold pitcher of imported beer",0
B3              byte        "Dr. Smith thanks you profusely for your "
                byte        "good taste in brews.",cr,lf
                byte        "He then invites you to the pub for a "
                byte        "round of pool and",cr,lf
                byte        "some heavy duty hob-nobbing, "
                byte        "CS Department style.",0
```

```
dseg            ends


cseg            segment     para public 'code'
                assume      ds:dseg


; SetNoun-      Copies the value in SI (the matchparm parameter) to the
;               NOUN variable.

SetNoun         proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     Noun, si
                mov     ax, di
                stc
                pop     ds
                ret
SetNoun         endp


; SetVerb-      Copies the value in SI (the matchparm parameter) to the
;               VERB variable.

SetVerb         proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     Verb, si
                mov     ax, di
                stc
                pop     ds
                ret
SetVerb         endp

; SetPtr-       Copies the value in SI (the matchparm parameter) to the
;               NOUNPTR variable.

SetPtr          proc    far
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     NounPtr, si
                mov     ax, di
                stc
                pop     ds
                ret
SetPtr          endp

; CheckPresence-
;               BX points at an item. DI points at an item list. This
;               routine checks to see if that item is present in the
;               item list. Returns Carry set if item was found,
;               clear if not found.

CheckPresence   proc

; MaxWeight is an assembly-time adjustable constant that determines
; how many objects the user can carry, or can be in a room, at one
; time. The following repeat macro emits "MaxWeight" compare and
; branch sequences to test each item pointed at by DS:DI.

ItemCnt         =       0
                repeat  MaxWeight
                cmp     bx, [di+ItemCnt]
                je      GotIt

ItemCnt         =       ItemCnt+2
                endm
```

```
                clc
                ret

GotIt:          stc
                ret
CheckPresence   endp

; RemoveItem-   BX contains a pointer to an item. DI contains a pointer
;               to an item list which contains that item. This routine
;               searches the item list and removes that item from the
;               list. To remove an item from the list, we need only
;               store a zero (NULL) over the top of its pointer entry
;               in the list.

RemoveItem      proc

; Once again, we use the repeat macro to automatically generate a chain
; of compare, branch, and remove code sequences for each possible item
; in the list.

ItemCnt         =       0
                repeat  MaxWeight
                local   NotThisOne
                cmp     bx, [di+ItemCnt]
                jne     NotThisOne
                mov     word ptr [di+ItemCnt], NULL
                ret
NotThisOne:
ItemCnt         =       ItemCnt+2
                endm

                ret
RemoveItem      endp


; InsertItem-   BX contains a pointer to an item, DI contains a pointer to
;               and item list. This routine searches through the list for
;               the first empty spot and copies the value in BX to that point.
;               It returns the carry set if it succeeds. It returns the
;               carry clear if there are no empty spots available.

InsertItem      proc

ItemCnt         =       0
                repeat  MaxWeight
                local   NotThisOne
                cmp     word ptr [di+ItemCnt], 0
                jne     NotThisOne
                mov     [di+ItemCnt], bx
                stc
                ret
NotThisOne:
ItemCnt         =       ItemCnt+2
                endm

                clc
                ret
InsertItem      endp

; LongDesc- Long description of an item.
; DI points at an item - print the long description of it.

LongDesc        proc
                push    di
                test    di, di
                jz      NoDescription
                mov     di, [di].item.LongDesc
                puts
                putcr
```

```
NoDescription:  pop     di
                ret
LongDesc        endp


; ShortDesc- Print the short description of an object.
; DI points at an item (possibly NULL). Print the short description for it.

ShortDesc       proc
                push    di
                test    di, di
                jz      NoDescription
                mov     di, [di].item.ShortDesc
                puts
                putcr
NoDescription:  pop     di
                ret
ShortDesc       endp

; Describe:     "CurRoom" points at the current room. Describe it and its
;               contents.

Describe        proc
                push    es
                push    bx
                push    di
                mov     di, ds
                mov     es, di

                mov     bx, CurRoom
                mov     di, [bx].room.Description
                print
                byte    "You are currently ",0
                puts
                putcr
                print
                byte    "Here you find the following:",cr,lf,0

; For each possible item in the room, print out the long description
; of that item. The repeat macro generates a code sequence for each
; possible item that could be in this room.

ItemCnt         =       0
                repeat  MaxWeight
                mov     di, [bx].room.ItemList[ItemCnt]
                call    LongDesc

ItemCnt         =       ItemCnt+2
                endm


                pop     di
                pop     bx
                pop     es
                ret
Describe        endp


; Here is the main program, that actually plays the game.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                print
                byte    cr,lf,lf,lf,lf,lf
                byte    "Welcome to ",'"MADVENTURE"',cr,lf
                byte    'If you need help, type the command "HELP"'
```

```
                byte    cr,lf,0

RoomLoop:       dec     CurScore    ;One point for each move.
                jnz     NotOverYet

; If they made too many moves without dropping anything properly, boot them
; out of the game.

                print
                byte    "WHOA! You lost! You get to join the legions of "
                byte    "the totally lame",cr,lf
                byte    'who have failed at "MADVENTURE"',cr,lf,0
                jmp     Quit

; Okay, tell 'em where they are and get a new command from them.

NotOverYet:     putcr
                call    Describe
                print
                byte    cr,lf
                byte    "Command: ",0
                lesi    InputLine
                gets
                strupr              ;Ignore case by converting to U.C.

; Okay, process the command. Note that we don't actually check to see
; if there is a properly formed sentence. Instead, we just look to see
; if any important keywords are on the line. If they are, the pattern
; matching routines load the appropriate values into the noun and verb
; variables (nouns: north=1, south=2, east=3, west=4, lime=5, beer=6,
; card=7, sign=8, program=9, homework=10, money=11, form=12, coupon=13;
; verbs: go=1, get=2, drop=3, inventory=4, quit=5, help=6).
;
; This code uses the noun and verb variables as indexes into a two
; dimensional array whose elements contain the address of the code
; to process the given command. If a given command does not make
; any sense (e.g., "go coupon") the entry in the table points at the
; bad command code.

                mov     Noun, 0
                mov     Verb, 0
                mov     NounPtr, 0

                ldxi    VerbPat
                xor     cx, cx
                match

                lesi    InputLine
                ldxi    NounPat
                xor     cx, cx
                match

; Okay, index into the command table and jump to the appropriate
; handler. Note that we will cheat and use a 14x8 array. There
; are really only seven verbs, not eight. But using eight makes
; things easier since it is easier to multiply by eight than seven.

                mov     si, CurRoom;The commands expect this here.

                mov     bx, Noun
                shl     bx, 3       ;Multiply by eight.
                add     bx, Verb
                shl     bx, 1       ;Multiply by two - word table.
                jmp     cseg:jmptbl[bx]

; The following table contains the noun x verb cross product.
; The verb values (in each row) are the following:
;
;       NONE      GO        GET DROP     INVNTRY    QUIT    HELP    unused
;        0         1         2  3         4          5       6       7
```

```
;
; There is one row for each noun (plus row zero, corresponding to no
; noun found on line).

jmptbl          word    Bad          ;No noun, no verb
                word    Bad          ;No noun, GO
                word    Bad          ;No noun, GET
                word    Bad          ;No noun, DROP
                word    DoInventory  ;No noun, INVENTORY
                word    QuitGame     ;No noun, QUIT
                word    DoHelp       ;No noun, HELP
                word    Bad          ;N/A

NorthCmds       word    Bad, GoNorth, Bad, Bad, Bad, Bad, Bad, Bad
SouthCmds       word    Bad, GoSouth, Bad, Bad, Bad, Bad, Bad, Bad
EastCmds        word    Bad, GoEast, Bad, Bad, Bad, Bad, Bad, Bad
WestCmds        word    Bad, GoWest, Bad, Bad, Bad, Bad, Bad, Bad
LimeCmds        word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
BeerCmds        word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
CardCmds        word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
SignCmds        word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
ProgramCmds     word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
HomeworkCmds    word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
MoneyCmds       word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
FormCmds        word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad
CouponCmds      word    Bad, Bad, GetItem, DropItem, Bad, Bad, Bad, Bad

; If the user enters a command we don't know how to process, print an
; appropriate error message down here.

Bad:            printf
                byte    "I'm sorry, I don't understand how to '%s'\n",0
                dword   InputLine
                jmp     NotOverYet


; Handle the movement commands here.
; Movements are easy, all we've got to do is fetch the NORTH, SOUTH,
; EAST, or WEST pointer from the current room's data structure and
; set the current room to that address. The only catch is that some
; moves are not legal. Such moves have a NULL (zero) in the direction
; field. A quick check for this case handles illegal moves.

GoNorth:        mov     si, [si].room.North
                jmp     MoveMe

GoSouth:        mov     si, [si].room.South
                jmp     MoveMe

GoEast:         mov     si, [si].room.East
                jmp     MoveMe

GoWest:         mov     si, [si].room.West
MoveMe:         test    si, si                  ;See if move allowed.
                jnz     SetCurRoom
                printf
                byte    "Sorry, you cannot go in this direction."
                byte    cr, lf, 0
                jmp     RoomLoop

SetCurRoom:     mov     CurRoom, si  ;Move to new room.
                jmp     RoomLoop

; Handle the GetItem command down here. At this time the user
; has entered GET and some noun that the player can pick up.
; First, we will make sure that item is in this room.
; Then we will check to make sure that picking up this object
; won't overload the player. If these two conditions are met,
; we'll transfer the object from the room to the player.
```

```
GetItem:        mov     bx, NounPtr  ;Ptr to item user wants.
                mov     si, CurRoom
                lea     di, [si].room.ItemList;Ptr to item list in di.
                call    CheckPresence;See if in room.
                jc      GotTheItem
                printf
                byte    "Sorry, that item is not available here."
                byte    cr, lf, 0
                jmp     RoomLoop

; Okay, see if picking up this object will overload the player.

GotTheItem:     mov     ax, [bx].Item.Weight
                add     ax, CurWeight
                cmp     ax, MaxWeight
                jbe     WeightOkay
                printf
                byte    "Sorry, you are already carrying too many items "
                byte    "to safely carry\nthat object\n",0
                jmp     RoomLoop

; Okay, everything's cool, transfer the object from the room to the user.

WeightOkay:     mov     CurWeight, ax;Save new weight.
                call    RemoveItem   ;Remove item from room.
                lea     di, ItemsOnHand;Ptr to player's list.
                call    InsertItem
                jmp     RoomLoop


; Handle dropped objects down here.

DropItem:       lea     di, ItemsOnHand;See if the user has
                mov     bx, NounPtr  ; this item on hand.
                call    CheckPresence
                jc      CanDropIt1
                printf
                byte    "You are not currently holding that item\n",0
                jmp     RoomLoop

; Okay, let's see if this is the magic room where this item is
; supposed to be dropped. If so, award the user some points for
; properly figuring this out.

CanDropIt1:     mov     ax, [bx].item.key
                cmp     ax, CurRoom
                jne     JustDropIt

; Okay, success! Print the winning message for this object.

                mov     di, [bx].item.WinDesc
                puts
                putcr

; Award the user some points.

                mov     ax, [bx].item.value
                add     CurScore, ax

; Since the user dropped it, they can carry more things now.

                mov     ax, [bx].item.Weight
                sub     CurWeight, ax

; Okay, take this from the user's list.

                lea     di, ItemsOnHand
                call    RemoveItem

; Keep track of how may objects the user has successfully dropped.
```

```
; When this counter hits zero, the game is over.

                dec     TotalCounter
                jnz     RoomLoop

                printf
                byte    "Well, you've found where everything goes "
                byte    "and your score is %d.\n"
                byte    "You might want to play again and see if "
                byte    "you can get a better score.\n",0
                dword   CurScore
                jmp     Quit

; If this isn't the room where this object belongs, just drop the thing
; off. If this object won't fit in this room, ignore the drop command.

JustDropIt:     mov     di, CurRoom
                lea     di, [di].room.ItemList
                call    InsertItem
                jc      DroppedItem
                printf
                byte    "There is insufficient room to leave "
                byte    "that item here.\n",0
                jmp     RoomLoop

; If they can drop it, do so. Don't forget we've just unburdened the
; user so we need to deduct the weight of this object from what the
; user is currently carrying.

DroppedItem:    lea     di, ItemsOnHand
                call    RemoveItem
                mov     ax, [bx].item.Weight
                sub     CurWeight, ax
                jmp     RoomLoop

; If the user enters the INVENTORY command, print out the objects on hand

DoInventory:    printf
                byte    "You currently have the following items in your "
                byte    "possession:",cr,lf,0
                mov     di, ItemsOnHand[0]
                call    ShortDesc
                mov     di, ItemsOnHand[2]
                call    ShortDesc
                mov     di, ItemsOnHand[4]
                call    ShortDesc
                mov     di, ItemsOnHand[6]
                call    ShortDesc
                printf
                byte    "\nCurrent score: %d\n"
                byte    "Carrying ability: %d/4\n\n",0
                dword   CurScore,CurWeight
                inc     CurScore        ;This command is free.
                jmp     RoomLoop


; If the user requests help, provide it here.

DoHelp:         printf
                byte    "List of commands:",cr,lf,lf
                byte    "GO {NORTH, EAST, WEST, SOUTH}",cr,lf
                byte    "{GET, DROP} {LIME, BEER, CARD, SIGN, PROGRAM, "
                byte    "HOMEWORK, MONEY, FORM, COUPON}",cr,lf
                byte    "SHOW INVENTORY",cr,lf
                byte    "QUIT GAME",cr,lf
                byte    "HELP ME",cr,lf,lf
                byte    "Each command costs you one point.",cr,lf
                byte    "You accumulate points by picking up objects and "
                byte    "dropping them in their",cr,lf
                byte    " appropriate locations.",cr,lf
```

```
                byte    "If you drop an item in its proper location, it "
                byte    "disappears from the game.",cr,lf
                byte    "The game is over if your score drops to zero or "
                byte    "you properly place",cr,lf
                byte    " all items.",cr,lf
                byte    0
                jmp     RoomLoop


; If they quit prematurely, let 'em know what a wimp they are!

QuitGame:       printf
                byte    "So long, your score is %d and there are "
                byte    "still %d objects unplaced\n",0
                dword   CurScore, TotalCounter

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

16.9 Laboratory Exercises

Programming with the Standard Library Pattern Matching routines doubles the complexity. Not only must you deal with the complexities of 80x86 assembly language, you must also deal with the complexities of the pattern matching paradigm, a programming language in its own right. While you can use a program like CodeView to track down problems in an assembly language program, no such debugger exists for "programs" you write with the Standard Library's pattern matching "language." Although the pattern matching routines are written in assembly language, attempting to trace through a pattern using CodeView will not be very enlightening. In this laboratory exercise, you will learn how to develop some rudimentary tools to help debug pattern matching programs.

16.9.1 Checking for Stack Overflow (Infinite Loops)

One common problem in pattern matching programs is the possibility of an infinite loop occurring in the pattern. This might occur, for example, if you have a left recursive production. Unfortunately, tracking down such loops in a pattern is very tedious, even with the help of a debugger like CodeView. Fortunately, there is a very simple change you can make to a program that uses patterns that will abort the program an warn you if infinite recursion exists.

Infinite recursion in a pattern occurs when sl_Match2 continuously calls itself without ever returning. This overflows the stack and causes the program to crash. There is a very easy change you can make to your programs to check for stack overflow:

- In patterns where you would normally call sl_Match2, call MatchPat instead.
- Include the following statements near the beginning of your program (before any patterns):

```
DEBUG           =       0           ;Define for debugging.

                ifdef   DEBUG
```

```
MatchPat        textequ  <MatchSP>
                else
MatchPat        textequ  <sl_Match2>
                endif
```

If you define the DEBUG symbol, your patterns will call the MatchSP procedure, otherwise they will call the sl_Match2 procedure. During testing, define the DEBUG symbol.

- Insert the following procedure somewhere in your program:

```
MatchSP         proc    far
                cmp      sp, offset StkOvrfl
                jbe      AbortPgm
                jmp      sl_Match2

AbortPgm:       print
                byte    cr,lf,lf
                byte     "Error: Stack overflow in MatchSP routine.",cr,lf,0
                ExitPgm
MatchSP         endp
```

This code sandwiches itself between your pattern and the sl_Match2 routine. It checks the stack pointer (sp) to see if it has dropped below a minimally acceptable point in the stack segment. If not, it continues execution by jumping to the sl_Match2 routine; otherwise it aborts program execution with an error message.

- The final change to your program is to modify the stack segment so that it looks like the following:

```
sseg            segment  para stack 'stack'
                word     64 dup (?)             ;Buffer for stack overflow
StkOvrfl        word     ?                      ;Stack overflow if drops
stk             db       1024 dup ("stack   ")  ; below StkOvrfl.
sseg            ends
```

After making these changes, your program will automatically stop with an error message if infinite recursion occurs since infinite recursion will most certainly cause a stack overflow[17].

The following code (Ex16_1a.asm on the companion CD-ROM) presents a simple calculator, similar to the calculator in the section “Evaluating Arithmetic Expressions” on page 948, although this calculator only supports addition. As noted in the comments appearing in this program, the pattern for the expression parser has a serious flaw – it uses a left recursive production. This will most certainly cause an infinite loop and a stack overflow. **For your lab report:** Run this program with and without the DEBUG symbol defined (i.e., comment out the definition for one run). Describe what happens.

```
; EX16_1a.asm
;
; A simple floating point calculator that demonstrates the use of the
; UCR Standard Library pattern matching routines.  Note that this
; program requires an FPU.

                .xlist
                .386
                .387
                option      segment:use16
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list
```

17. This code will also abort your program if you use too much stack space without infinite recursion. A problem in its own right.

```
; If the symbol "DEBUG" is defined, then call the MatchSP routine
; to do stack overflow checking.  If "DEBUG" is not defined, just
; call the sl_Match2 routine directly.

DEBUG           =       0           ;Define for debugging.

                ifdef   DEBUG
MatchPat        textequ <MatchSP>
                else
MatchPat        textequ <sl_Match2>
                endif

dseg            segment para public 'data'

; The following is a temporary used when converting a floating point
; string to a 64 bit real value.

CurValue        real8   0.0

; A Test String:

TestStr         byte    "5+2-(3-1)",0

; Grammar for simple infix -> postfix translation operation:
; Semantic rules appear in braces.
;
; NOTE: This code has a serious problem.  The first production
; is left recursive and will generate an infinite loop.
;
; E -> E+T {print result} | T {print result}
; T -> <constant> {fld constant} | (E)
;
;
; UCR Standard Library Pattern that handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

Expression      pattern {MatchPat,E,,EndOfString}
EndOfString     pattern {EOS}


; An "E" item consists of an "E" item optionally followed by "+" or "-"
; and a "T" item (E -> E+T | T):

E               pattern {MatchPat, E,T,Eplus}
Eplus           pattern {MatchChar, '+', T, epPlus}
epPlus          pattern {DoFadd}


; A "T" item is either a floating point constant or "(" followed by
; an "E" item followed by ")".
;
; The regular expression for a floating point constant is
;
;       [0-9]+ ( "." [0-9]* | ) ( ((e|E) (+|-| ) [0-9]+) | )
;
; Note: the pattern "Const" matches exactly the characters specified
;       by the above regular expression.  It is the pattern the calc-
;       ulator grabs when converting a string to a floating point number.


Const           pattern {MatchPat, ConstStr, 0, FLDConst}
ConstStr        pattern {MatchPat, DoDigits, 0, Const2}
Const2          pattern {matchchar, '.', Const4, Const3}
Const3          pattern {MatchPat, DoDigits, Const4, Const4}
Const4          pattern {matchchar, 'e', const5, const6}
Const5          pattern {matchchar, 'E', Succeed, const6}
Const6          pattern {matchchar, '+', const7, const8}
Const7          pattern {matchchar, '-', const8, const8}
```

```
Const8          pattern  {MatchPat, DoDigits}

FldConst        pattern  {PushValue}

; DoDigits handles the regular expression [0-9]+

DoDigits        pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern  {Spancset, Digits}

; The S production handles constants or an expression in parentheses.

T               pattern  {MatchChar, '(', Const, IntE}
IntE            pattern  {MatchPat, E, 0, CloseParen}
CloseParen      pattern  {MatchChar, ')'}


; The Succeed pattern always succeeds.

Succeed         pattern  {DoSucceed}


; We use digits from the UCR Standard Library cset standard sets.

                include  stdsets.a

dseg            ends



cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Debugging feature #1:
; This is a special version of sl_Match2 that checks for
; stack overflow.  Stack overflow occurs whenever there
; is an infinite loop (i.e., left recursion) in a pattern.

MatchSP         proc    far
                cmp      sp, offset StkOvrfl
                jbe      AbortPgm
                jmp      sl_Match2

AbortPgm:       print
                byte    cr,lf,lf
                byte     "Error: Stack overflow in MatchSP routine.",cr,lf,0
                ExitPgm
MatchSP         endp

; DoSucceed matches the empty string.  In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed       proc     far
                mov      ax, di
                stc
                ret
DoSucceed       endp

; DoFadd - Adds the two items on the top of the FPU stack.

DoFadd          proc     far
                faddp    st(1), st
                mov      ax, di                 ;Required by sl_Match
                stc                             ;Always succeed.
                ret
DoFadd          endp


; PushValue-    We've just matched a string that corresponds to a
;               floating point constant.  Convert it to a floating
```

```
;               point value and push that value onto the FPU stack.

PushValue       proc    far
                push    ds
                push    es
                pusha
                mov     ax, dseg
                mov     ds, ax

                lesi    Const       ;FP val matched by this pat.
                patgrab             ;Get a copy of the string.
                atof                ;Convert to real.
                free                ;Return mem used by patgrab.
                lesi    CurValue    ;Copy floating point accumulator
                sdfpa               ; to a local variable and then
                fld     CurValue    ; copy that value to the FPU stk.

                popa
                mov     ax, di
                pop     es
                pop     ds
                stc
                ret
PushValue       endp

; The main program tests the expression evaluator.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                finit               ;Be sure to do this!
                fwait

                lesi    TestStr
                puts                ;Print the expression

                ldxi    Expression
                xor     cx, cx
                match
                jc      GoodVal
                printff
                byte    " is an illegal expression",cr,lf,0
                ret

GoodVal:        fstp    CurValue
                printff
                byte    " = %12.6ge\n",0
                dword   CurValue

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment para stack 'stack'
                word    64 dup (?)  ;Buffer for stack overflow
StkOvrfl        word    ?                   ;Stack overflow if drops
stk             db      1024 dup ("stack   "); below StkOvrfl.
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

16.9.2 Printing Diagnostic Messages from a Pattern

When there is no other debugging method available, you can always use print statements to help track down problems in your patterns. If your program calls pattern matching functions in your own code (like the DoFAdd, DoSucceed, and PushValue procedures in the code above), you can easily insert print or printf statements in these functions that will print an appropriate message when they execute. Unfortunately, a problem may develop in a portion of a pattern that does not call any local pattern matching functions, so inserting print statements within an existing (local) pattern matching function might not help. To solve this problem, all you need to do is insert a call to a local pattern matching function in the patterns you suspect have a problem.

Rather than make up a specific local pattern to print an individual message, a better solution is to write a generic pattern matching function whose whole purpose is to display a message. The following PatPrint function does exactly this:

```
; PatPrint- A debugging aid.  This "Pattern matching function" prints
; the string that DS:SI points at.

PatPrint        proc    far
                push    es
                push    di
                mov     di, ds
                mov     es, di
                mov     di, si
                puts
                mov     ax, di
                pop     di
                pop     es
                stc
                ret
PatPrint        endp
```

From “Constructing Patterns for the MATCH Routine” on page 933, you will note that the pattern matching system passes the value of the MatchParm parameter to a pattern matching function in the ds:si register pair. The PatPrint function prints the string that ds:si points at (by moving ds:si to es:di and calling puts).

The following code (Ex16_1b.asm on the companion CD-ROM) demonstrates how to insert calls to PatPrint within your patterns to print out data to help you track down problems in your patterns. **For your lab report:** run this program and describe its output in your report. Describe how this output can help you track down the problem with this program. Modify the grammar to match the grammar in the corresponding sample program (see “Evaluating Arithmetic Expressions” on page 948) while still printing out each production that this program processes. Run the result and include the output in your lab report.

```
; EX16_1a.asm
;
; A simple floating point calculator that demonstrates the use of the
; UCR Standard Library pattern matching routines.  Note that this
; program requires an FPU.

                .xlist
                .386
                .387
                option      segment:use16
                include     stdlib.a
                includelib  stdlib.lib
                matchfuncs
                .list

; If the symbol "DEBUG" is defined, then call the MatchSP routine
; to do stack overflow checking.  If "DEBUG" is not defined, just
; call the sl_Match2 routine directly.
```

```
DEBUG           =       0           ;Define for debugging.

                ifdef   DEBUG
MatchPat        textequ <MatchSP>
                else
MatchPat        textequ <sl_Match2>
                endif

dseg            segment para public 'data'

; The following is a temporary used when converting a floating point
; string to a 64 bit real value.

CurValue        real8   0.0


; A Test String:

TestStr         byte    "5+2-(3-1)",0


; Grammar for simple infix -> postfix translation operation:
; Semantic rules appear in braces.
;
; NOTE: This code has a serious problem.  The first production
; is left recursive and will generate an infinite loop.
;
; E -> E+T {print result} | T {print result}
; T -> <constant> {fld constant} | (E)
;
; UCR Standard Library Pattern that handles the grammar above:

; An expression consists of an "E" item followed by the end of the string:

Expression      pattern {MatchPat,E,,EndOfString}
EndOfString     pattern {EOS}


; An "E" item consists of an "E" item optionally followed by "+" or "-"
; and a "T" item (E -> E+T | T):

E               pattern {PatPrint,EMsg,,E2}
EMsg            byte    "E->E+T | T",cr,lf,0

E2              pattern {MatchPat, E,T,Eplus}
Eplus           pattern {MatchChar, '+', T, epPlus}
epPlus          pattern {DoFadd,,,E3}
E3              pattern {PatPrint,EMsg3}
EMsg3           byte    "E->E+T",cr,lf,0


; A "T" item is either a floating point constant or "(" followed by
; an "E" item followed by ")".
;
; The regular expression for a floating point constant is
;
;       [0-9]+ ( "." [0-9]* | ) ( ((e|E) (+|-| ) [0-9]+) | )
;
; Note: the pattern "Const" matches exactly the characters specified
;       by the above regular expression.  It is the pattern the calc-
;       ulator grabs when converting a string to a floating point number.

Const           pattern {MatchPat, ConstStr, 0, FLDConst}
ConstStr        pattern {MatchPat, DoDigits, 0, Const2}
Const2          pattern {matchchar, '.', Const4, Const3}
Const3          pattern {MatchPat, DoDigits, Const4, Const4}
Const4          pattern {matchchar, 'e', const5, const6}
Const5          pattern {matchchar, 'E', Succeed, const6}
Const6          pattern {matchchar, '+', const7, const8}
```

```
Const7          pattern  {matchchar, '-', const8, const8}
Const8          pattern  {MatchPat, DoDigits}

FldConst        pattern  {PushValue,,,ConstMsg}
ConstMsg        pattern  {PatPrint,CMsg}
CMsg            byte     "T->const",cr,lf,0

; DoDigits handles the regular expression [0-9]+

DoDigits        pattern  {Anycset, Digits, 0, SpanDigits}
SpanDigits      pattern  {Spancset, Digits}

; The S production handles constants or an expression in parentheses.

T               pattern  {PatPrint,TMsg,,T2}
TMsg            byte     "T->(E) | const",cr,lf,0

T2              pattern  {MatchChar, '(', Const, IntE}
IntE            pattern  {MatchPat, E, 0, CloseParen}
CloseParen      pattern  {MatchChar, ')',,T3}

T3              pattern  {PatPrint,TMsg3}
TMsg3           byte     "T->(E)",cr,lf,0

; The Succeed pattern always succeeds.

Succeed         pattern  {DoSucceed}

; We use digits from the UCR Standard Library cset standard sets.

                include  stdsets.a

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; Debugging feature #1:
; This is a special version of sl_Match2 that checks for
; stack overflow.  Stack overflow occurs whenever there
; is an infinite loop (i.e., left recursion) in a pattern.

MatchSP         proc    far
                cmp      sp, offset StkOvrfl
                jbe      AbortPgm
                jmp      sl_Match2

AbortPgm:       print
                byte     cr,lf,lf
                byte     "Error: Stack overflow in MatchSP routine.",cr,lf,0
                ExitPgm
MatchSP         endp

; PatPrint- A debugging aid.  This "Pattern matching function" prints
; the string that DS:SI points at.

PatPrint        proc     far
                push     es
                push     di
                mov      di, ds
                mov      es, di
                mov      di, si
                puts
                mov      ax, di
                pop      di
                pop      es
                stc
                ret
PatPrint        endp
```

```
; DoSucceed matches the empty string.  In other words, it matches anything
; and always returns success without eating any characters from the input
; string.

DoSucceed       proc    far
                mov     ax, di
                stc
                ret
DoSucceed       endp

; DoFadd - Adds the two items on the top of the FPU stack.

DoFadd          proc    far
                faddp   st(1), st
                mov     ax, di                  ;Required by sl_Match
                stc                             ;Always succeed.
                ret
DoFadd          endp

; PushValue-    We've just matched a string that corresponds to a
;               floating point constant.  Convert it to a floating
;               point value and push that value onto the FPU stack.

PushValue       proc    far
                push    ds
                push    es
                pusha
                mov     ax, dseg
                mov     ds, ax

                lesi    Const           ;FP val matched by this pat.
                patgrab                 ;Get a copy of the string.
                atof                    ;Convert to real.
                free                    ;Return mem used by patgrab.
                lesi    CurValue        ;Copy floating point accumulator
                sdfpa                   ; to a local variable and then
                fld     CurValue        ; copy that value to the FPU stk.

                popa
                mov     ax, di
                pop     es
                pop     ds
                stc
                ret
PushValue       endp



; The main program tests the expression evaluator.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                finit                   ;Be sure to do this!
                fwait

                lesi    TestStr
                puts                    ;Print the expression

                ldxi    Expression
                xor     cx, cx
                match
                jc      GoodVal
                printff
                byte    " is an illegal expression",cr,lf,0
                ret
```

```
GoodVal:fstp    CurValue
                printff
                byte    " = %12.6ge\n",0
                dword   CurValue

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment para stack 'stack'
                word    64 dup (?)              ;Buffer for stack overflow
StkOvrfl        word    ?                       ;Stack overflow if drops
stk             db      1024 dup ("stack   ")   ; below StkOvrfl.
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

16.10 Programming Projects

1) Modify the program in Section 16.8.3 (Arith2.asm on the companion CD-ROM) so that it includes some common trigonometric operations (sin, cos, tan, etc.). See the chapter on floating point arithmetic to see how to compute these functions. The syntax for the functions should be similar to “sin(E)” where “E” represents an arbitrary expression.

2) Modify the (English numeric input problem in Section 16.8.1 to handle negative numbers. The pattern should allow the use of the prefixes “negative” or “minus” to denote a negative number.

3) Modify the (English) numeric input problem in Section 16.8.1 to handle four byte unsigned integers.

4) Write your own “Adventure” game based on the programming techniques found in the “Madventure” game in Section 16.8.5.

5) Write a “tiny assembler” for the modern version of the x86 processor using the techniques found in Section 16.8.4.

6) Write a simple “DOS Shell” program that reads a line of text from the user and processes valid DOS commands found on that line. Handle at least the DEL, RENAME, TYPE, and COPY commands. See “MS-DOS, PC-BIOS, and File I/O” on page 699 for information concerning the implementation of these DOS commands.

16.11 Summary

This has certainly been a long chapter. The general topic of pattern matching receives insufficient attention in most textbooks. In fact, you rarely see more than a dozen or so pages dedicated to it outside of automata theory texts, compiler texts, or texts covering pattern matching languages like Icon or SNOBOL4. That is one of the main reasons this chapter is extensive, to help cover the paucity of information available elsewhere. However, there is another reason for the length of this chapter and, especially, the number of lines of code appearing in this chapter – to demonstrate how easy it is to develop certain classes of programs using pattern matching techniques. Could you imagine having to write a program like Madventure using standard C or Pascal programming techniques? The resulting program would probably be longer than the assembly version appearing in this chapter! If you are not impressed with the power of pattern matching, you should probably reread this chapter. It is very surprising how few programmers truly understand the theory of pattern matching; especially considering how many program use, or could benefit from, pattern matching techniques.

This chapter begins by discussing the theory behind pattern matching. It discusses simple patterns, known as *regular languages*, and describes how to design *nondeterministic* and *deterministic finite state automata* – the functions that match patterns described by *regular expressions*. This chapter also describes how to convert NFAs and DFAs into assembly language programs. For the details, see

- "An Introduction to Formal Language (Automata) Theory" on page 883
- "Machines vs. Languages" on page 883
- "Regular Languages" on page 884
- "Regular Expressions" on page 885
- "Nondeterministic Finite State Automata (NFAs)" on page 887
- "Converting Regular Expressions to NFAs" on page 888
- "Converting an NFA to Assembly Language" on page 890
- "Deterministic Finite State Automata (DFAs)" on page 893
- "Converting a DFA to Assembly Language" on page 895

Although the regular languages are probably the most commonly processed patterns in modern pattern matching programs, they are also only a small subset of the possible types of patterns you can process in a program. The *context free languages* include all the regular languages as a subset and introduce many types of patterns that are not regular. To represent a context free language, we often use a *context free grammar*. A CFG contains a set of expressions known as *productions*. This set of productions, a set of *nonterminal symbols*, a set of *terminal symbols*, and a special nonterminal, the *starting symbol*, provide the basis for converting powerful patterns into a programming language.

In this chapter, we've covered a special set of the context free grammars known as LL(1) grammars. To properly encode a CFG as an assembly language program, you must first convert the grammar to an LL(1) grammar. This encoding yields a *recursive descent predictive parser*. Two primary steps required before converting a grammar to a program that recognizes strings in the context free language is to *eliminate left recursion* from the grammar and *left factor* the grammar. After these two steps, it is relatively easy to convert a CFG to an assembly language program.

For more information on CFGs, see

- "Context Free Languages" on page 900
- "Eliminating Left Recursion and Left Factoring CFGs" on page 903
- "Converting CFGs to Assembly Language" on page 905
- "Some Final Comments on CFGs" on page 912

Sometimes it is easier to deal with regular expressions rather than context free grammars. Since CFGs are more powerful than regular expressions, this text generally adopts grammars whereever possible However, regular expressions are generally easier to work with (for simple patterns), especially in the early stages of development. Sooner or later, though, you may need to convert a regular expression to a CFG so you can combine it with other components of the grammar. This is very easy to do and there is a simple algorithm to convert REs to CFGs. For more details, see

- "Converting REs to CFGs" on page 905

Although converting CFGs to assembly language is a straightforward process, it is very tedious. The UCR Standard Library includes a set of pattern matching routines that completely eliminate this tedium and provide many additional capabilities as well (such as automatic backtracking, allowing you to encode grammars that are not LL(1)). The pattern matching package in the Standard Library is probably the most novel and powerful set of routines available therein. You should definitely investigate the use of these routines, they can save you considerable time. For more information, see

- "The UCR Standard Library Pattern Matching Routines" on page 913
- "The Standard Library Pattern Matching Functions" on page 914

One neat feature the Standard Library provides is your ability to write customized pattern matching functions. In addition to letting you provide pattern matching facilities

missing from the library, these pattern matching functions let you add *semantic rules* to your grammars. For all the details, see

- "Designing Your Own Pattern Matching Routines" on page 922
- "Extracting Substrings from Matched Patterns" on page 925
- "Semantic Rules and Actions" on page 929

Although the UCR Standard Library provides a powerful set of pattern matching routines, its richness may be its primary drawback. Those who encounter the Standard Library's pattern matching routines for the first time may be overwhelmed, especially when attempting to reconcile the material in the section on context free grammars with the Standard Library patterns. Fortunately, there is a straightforward, if inefficient, way to translate CFGs into Standard Library patterns. This technique is outlined in

- "Constructing Patterns for the MATCH Routine" on page 933

Although pattern matching is a very powerful paradigm that most programmers should familiarize themselves with, most people have a hard time seeing the applications when they first encounter pattern matching. Therefore, this chapter concludes with some very complete programs that demonstrate pattern matching in action. These examples appear in the section:

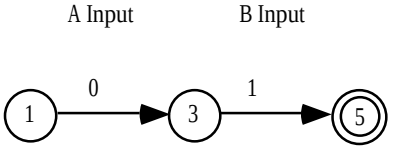
- "Some Sample Pattern Matching Applications" on page 935

16.12 Questions

1) Assume that you have two inputs that are either zero or one. Create a DFA to implement the following logic functions (assume that arriving in a final state is equivalent to being true, if you wind up in a non-accepting state you return false)

a) OR b) XOR c) NAND d) NOR

e) Equals (XNOR) f) AND



Example, A<B

2) If *r*, *s*, and *t* are regular expressions, what strings with the following regular expressions match?

a) r^* b) $r\ s$ c) r^+ d) $r \mid s$

3) Provide a regular expression for integers that allow commas every three digits as per U.S. syntax (e.g., for every three digits from the right of the number there must be exactly one comma). Do not allow misplaced commas.

4) Pascal real constants must have at least one digit before the decimal point. Provide a regular expression for FORTRAN real constants that does not have this restriction.

5) In many language systems (e.g., FORTRAN and C) there are two types of floating point numbers, single precision and double precision. Provide a regular expression for real numbers that allows the input of floating point numbers using any of the characters [dDeE] as the exponent symbol (d/D stands for double precision).

6) Provide an NFA that recognizes the mnemonics for the 886 instruction set.

7) Convert the NFA above into assembly language. Do not use the Standard Library pattern matching routines.

8) Repeat question (7) using the Standard Library pattern matching routines.

9) Create a DFA for Pascal identifiers.

10) Convert the above DFA to assembly code using straight assembly statements.

11) Convert the above DFA to assembly code using a state table with input classification. Describe the data in your classification table.

12) Eliminate left recursion from the following grammar:

```
Stmt    →   if expression then Stmt endif
        |   if expression then Stmt else Stmt endif
        |   Stmt ; Stmt
        |   ε
```

13) Left factor the grammar you produce in problem 12.

14) Convert the result from question (13) into assembly language without using the Standard Library pattern matching routines.

15) Convert the result from question (13) in assembly language using the Standard Library pattern matching routines.

16) Convert the regular expression obtained in question (3) to a set of productions for a context free grammar.

17) Why is the ARB matching function inefficient? Describe how the pattern (ARB "hello" ARB) would match the string "hello there".

18) Spancset matches zero or more occurrences of some characters in a character set. Write a pattern matching function, callable as the first field of the pattern data type, that matches one or more occurrences of some character (feel free to look at the sources for spancset).

19) Write the matchichar pattern matching function that matches an individual character regardless of case (feel free to look at the sources for matchchar).

20) Explain how to use a pattern matching function to implement a semantic rule.

21) How would you extract a substring from a matched pattern?

22) What are *parenthetical patterns*? How to you create them?

Interrupts, Traps, and Exceptions Chapter 17

The concept of an interrupt is something that has expanded in scope over the years. The 80x86 family has only added to the confusion surrounding interrupts by introducing the int (software interrupt) instruction. Indeed, different manufacturers have used terms like *exceptions, faults, aborts, traps,* and *interrupts* to describe the phenomena this chapter discusses. Unfortunately, there is no clear consensus as to the exact meaning of these terms. Different authors adopt different terms to their own use. While it is tempting to avoid the use of such misused terms altogether, for the purpose of discussion it would be nice to have a set of well defined terms we can use in this chapter. Therefore, we will pick three of the terms above, interrupts, traps, and exceptions, and define them. This chapter attempts to use the most common meanings for these terms, but don't be surprised to find other texts using them in different contexts.

On the 80x86, there are three types of events commonly known as interrupts: *traps*, *exceptions*, and *interrupts* (hardware interrupts). This chapter will describe each of these forms and discuss their support on the 80x86 CPUs and PC compatible machines.

Although the terms trap and exception are often used synonymously, we will use the term *trap* to denote a programmer initiated and expected transfer of control to a special handler routine. In many respects, a trap is nothing more than a specialized subroutine call. Many texts refer to traps as *software interrupts*. The 80x86 int instruction is the main vehicle for executing a trap. Note that traps are usually *unconditional*; that is, when you execute an int instruction, control *always* transfers to the procedure associated with the trap. Since traps execute via an explicit instruction, it is easy to determine exactly which instructions in a program will invoke a *trap handling* routine.

An exception is an automatically generated trap (coerced rather than requested) that occurs in response to some exceptional condition. Generally, there isn't a specific instruction associated with an exception[1], instead, an exception occurs in response to some degenerate behavior of normal 80x86 program execution. Examples of conditions that may *raise* (cause) an exception include executing a division instruction with a zero divisor, executing an illegal opcode, and a memory protection fault. Whenever such a condition occurs, the CPU immediately suspends execution of the current instruction and transfers control to an *exception handler* routine. This routine can decide how to handle the exceptional condition; it can attempt to rectify the problem or abort the program and print an appropriate error message. Although you do not generally execute a specific instruction to cause an exception, as with the software interrupts (traps), execution of some instruction is what causes an exception. For example, you only get a division error when executing a division instruction somewhere in a program.

Hardware interrupts, the third category that we will refer to simply as *interrupts*, are program control interruption based on an external hardware event (external to the CPU). These interrupts generally have nothing at all to do with the instructions currently executing; instead, some event, such as pressing a key on the keyboard or a time out on a timer chip, informs the CPU that a device needs some attention. The CPU interrupts the currently executing program, services the device, and then returns control back to the program.

An *interrupt service routine* is a procedure written specifically to handle a trap, exception, or interrupt. Although different phenomenon cause traps, exceptions, and interrupts, the structure of an interrupt service routine, or *ISR*, is approximately the same for each of these.

1. Although we will classify the into instruction in this category. This is an exception to this rule.

17.1 80x86 Interrupt Structure and Interrupt Service Routines (ISRs)

Despite the different causes of traps, exceptions, and interrupts, they share a common format for their handling routines. Of course, these interrupt service routines will perform different activities depending on the source of the invocation, but it is quite possible to write a single interrupt handling routine that processes traps, exceptions, and hardware interrupts. This is rarely done, but the structure of the 80x86 interrupt system allows this. This section will describe the 80x86's interrupt structure and how to write basic interrupt service routines for the 80x86 real mode interrupts.

The 80x86 chips allow up to 256 *vectored* interrupts. This means that you can have up to 256 different sources for an interrupt and the 80x86 will directly call the service routine for that interrupt without any software processing. This is in contrast to *nonvectored* interrupts that transfer control directly to a single interrupt service routine, regardless of the interrupt source.

The 80x86 provides a 256 entry *interrupt vector table* beginning at address 0:0 in memory. This is a 1K table containing 256 4-byte entries. Each entry in this table contains a segmented address that points at the interrupt service routine in memory. Generally, we will refer to interrupts by their index into this table, so interrupt zero's address (vector) is at memory location 0:0, interrupt one's vector is at address 0:4, interrupt two's vector is at address 0:8, etc.

When an interrupt occurs, regardless of source, the 80x86 does the following:

1) The CPU pushes the flags register onto the stack.

2) The CPU pushes a far return address (segment:offset) onto the stack, segment value first.

3) The CPU determines the cause of the interrupt (i.e., the interrupt number) and fetches the four byte interrupt vector from address 0:vector*4.

4) The CPU transfers control to the routine specified by the interrupt vector table entry.

After the completion of these steps, the interrupt service routine takes control. When the interrupt service routine wants to return control, it must execute an iret (interrupt return) instruction. The interrupt return pops the far return address and the flags off the stack. Note that executing a far return is insufficient since that would leave the flags on the stack.

There is one minor difference between how the 80x86 processes hardware interrupts and other types of interrupts – upon entry into the hardware interrupt service routine, the 80x86 disables further hardware interrupts by clearing the interrupt flag. Traps and exceptions do not do this. If you want to disallow further hardware interrupts within a trap or exception handler, you must explicitly clear the interrupt flag with a cli instruction. Conversely, if you want to allow interrupts within a hardware interrupt service routine, you must explicitly turn them back on with an sti instruction. Note that the 80x86's interrupt disable flag only affects hardware interrupts. Clearing the interrupt flag will not prevent the execution of a trap or exception.

ISRs are written like almost any other assembly language procedure except that they return with an iret instruction rather than ret. Although the distance of the ISR procedure (near vs. far) is usually of no significance, you should make all ISRs *far* procedures. This will make programming easier if you decide to call an ISR directly rather than using the normal interrupt handling mechanism.

Exceptions and hardware interrupts ISRs have a very special restriction: they must *preserve the state of the CPU*. In particular, these ISRs must preserve all registers they modify. Consider the following extremely simple ISR:

```
SimpleISR       proc    far
                mov     ax, 0
                iret
SimpleISR       endp
```

This ISR obviously does *not* preserve the machine state; it explicitly disturbs the value in ax and then returns from the interrupt. Suppose you were executing the following code segment when a hardware interrupt transferred control to the above ISR:

```
                mov     ax, 5
                add     ax, 2

; Suppose the interrupt occurs here.

                puti
                  .
                  .
                  .
```

The interrupt service routine would set the ax register to zero and your program would print zero rather than the value five. Worse yet, hardware interrupts are generally *asynchronous*, meaning they can occur at any time and rarely do they occur at the same spot in a program. Therefore, the code sequence above would print seven most of the time; once in a great while it might print zero or two (it will print two if the interrupt occurs between the mov ax, 5 and add ax, 2 instructions). Bugs in hardware interrupt service routines are very difficult to find, because such bugs often affect the execution of unrelated code.

The solution to this problem, of course, is to make sure you preserve all registers you use in the interrupt service routine for hardware interrupts and exceptions. Since trap calls are explicit, the rules for preserving the state of the machine in such programs is identical to that for procedures.

Writing an ISR is only the first step to implementing an interrupt handler. You must also initialize the interrupt vector table entry with the address of your ISR. There are two common ways to accomplish this – store the address directly in the interrupt vector table or call DOS and let DOS do the job for you.

Storing the address yourself is an easy task. All you need to do is load a segment register with zero (since the interrupt vector table is in segment zero) and store the four byte address at the appropriate offset within that segment. The following code sequence initializes the entry for interrupt 255 with the address of the SimpleISR routine presented earlier:

```
                mov     ax, 0
                mov     es, ax
                pushf
                cli
                mov     word ptr es:[0ffh*4], offset SimpleISR
                mov     word ptr es:[0ffh*4 + 2], seg SimpleISR
                popf
```

Note how this code turns off the interrupts while changing the interrupt vector table. This is important if you are patching a hardware interrupt vector because it wouldn't do for the interrupt to occur between the last two mov instructions above; at that point the interrupt vector is in an inconsistent state and invoking the interrupt at that point would transfer control to the offset of SimpleISR and the segment of the previous interrupt 0FFh handler. This, of course, would be a disaster. The instructions that turn off the interrupts while patching the vector are unnecessary if you are patching in the address of a trap or exception handler[2].

Perhaps a better way to initialize an interrupt vector is to use DOS' *Set Interrupt Vector* call. Calling DOS (see "MS-DOS, PC-BIOS, and File I/O" on page 699) with ah equal to 25h provides this function. This call expects an interrupt number in the al register and the address of the interrupt service routine in ds:dx. The call to MS-DOS that would accomplish the same thing as the code above is

2. Strictly speaking, this code sequence does not require the pushf, cli, and popf instructions because interrupt 255 does not correspond to any hardware interrupt on a typical PC machine. However, it is important to provide this example so you're aware of the problem.

```
                mov     ax, 25ffh                       ;AH=25h, AL=0FFh.
                mov     dx, seg SimpleISR               ;Load DS:DX with
                mov     ds, dx                          ; address of ISR
                lea     dx, SimpleISR
                int     21h                             ;Call DOS
                mov     ax, dseg                        ;Restore DS so it
                mov     ds, ax                          ; points back at DSEG.
```

Although this code sequence is a little more complex than poking the data directly into the interrupt vector table, it is safer. Many programs monitor changes made to the interrupt vector table through DOS. If you call DOS to change an interrupt vector table entry, those programs will become aware of your changes. If you circumvent DOS, those programs may not find out that you've patched in your own interrupt and could malfunction.

Generally, it is a very bad idea to patch the interrupt vector table and not restore the original entry after your program terminates. Well behaved programs always save the previous value of an interrupt vector table entry and restore this value before termination. The following code sequences demonstrate how to do this. First, by patching the table directly:

```
                mov     ax, 0
                mov     es, ax

; Save the current entry in the dword variable IntVectSave:

                mov     ax, es:[IntNumber*4]
                mov     word ptr IntVectSave, ax
                mov     ax, es:[IntNumber*4 + 2]
                mov     word ptr IntVectSave+2, ax

; Patch the interrupt vector table with the address of our ISR

                pushf                   ;Required if this is a hw interrupt.
                cli                     ; "       "   "   " "  "     "

                mov     word ptr es:[IntNumber*4], offset OurISR
                mov     word ptr es:[IntNumber*4+2], seg OurISR

                popf                    ;Required if this is a hw interrupt.

; Okay, do whatever it is that this program is supposed to do:

                .
                .
                .

; Restore the interrupt vector entries before quitting:

                mov     ax, 0
                mov     es, ax

                pushf                   ;Required if this is a hw interrupt.
                cli                     ;   "      "  "    "   "     "

                mov     ax, word ptr IntVectSave
                mov     es:[IntNumber*4], ax
                mov     ax, word ptr IntVectSave+2
                mov     es:[IntNumber*4 + 2], ax

                popf                    ;Required if this is a hw interrupt.
                .
                .
                .
```

If you would prefer to call DOS to save and restore the interrupt vector table entries, you can obtain the address of an existing interrupt table entry using the DOS *Get Interrupt Vector* call. This call, with ah=35h, expects the interrupt number in al; it returns the existing vector for that interrupt in the es:bx registers. Sample code that preserves the interrupt vector using DOS is

```
; Save the current entry in the dword variable IntVectSave:

                mov     ax, 3500h + IntNumber           ;AH=35h, AL=Int #.
                int     21h
                mov     word ptr IntVectSave, bx
                mov     word ptr IntVectSave+2, es

; Patch the interrupt vector table with the address of our ISR

                mov     dx, seg OurISR
                mov     ds, dx
                lea     dx, OurISR
                mov     ax, 2500h + IntNumber           ;AH=25, AL=Int #.
                int     21h

; Okay, do whatever it is that this program is supposed to do:

                .
                .
                .

; Restore the interrupt vector entries before quitting:

                lds     bx, IntVectSave
                mov     ax, 2500h+IntNumber             ;AH=25, AL=Int #.
                int     21h
                .
                .
                .
```

17.2 Traps

A trap is a software-invoked interrupt. To execute a trap, you use the 80x86 int (software interrupt) instruction[3]. There are only two primary differences between a trap and an arbitrary far procedure call: the instruction you use to call the routine (int vs. call) and the fact that a trap pushes the flags on the stack so you must use the iret instruction to return from it. Otherwise, there really is no difference between a trap handler's code and the body of a typical far procedure.

The main purpose of a trap is to provide a fixed subroutine that various programs can call without having to actually know the run-time address. MS-DOS is the perfect example. The int 21h instruction is an example of a trap invocation. Your programs do not have to know the actual memory address of DOS' entry point to call DOS. Instead, DOS patches the interrupt 21h vector when it loads into memory. When you execute int 21h, the 80x86 automatically transfers control to DOS' entry point, whereever in memory that happens to be.

There is a long lists of support routines that use the trap mechanism to link application programs to themselves. DOS, BIOS, the mouse drivers, and Netware™ are a few examples. Generally, you would use a trap to call a *resident program* function. Resident programs (see "Resident Programs" on page 1025) load themselves into memory and remain resident once they terminate. By patching an interrupt vector to point at a subroutine within the resident code, other programs that run after the resident program terminates can call the resident subroutines by executing the appropriate int instruction.

Most resident programs do *not* use a separate interrupt vector entry for each function they provide. Instead, they usually patch a *single* interrupt vector and transfer control to an appropriate routine using a *function number* that the caller passes in a register. By convention, most resident programs expect the function number in the ah register. A typical trap handler would execute a case statement on the value in the ah register and transfer control to the appropriate handler function.

3. You can also simulate an int instruction by pushing the flags and executing a far call to the trap handler. We will consider this mechanism later on.

Since trap handlers are virtually identical to far procedures in terms of use, we will not discuss traps in any more detail here. However, the text chapter will explore this subject in greater depth when it discusses resident programs.

17.3 Exceptions

Exceptions occur (are *raised*) when an abnormal condition occurs during execution. There are fewer than eight possible exceptions on machines running in real mode. Protected mode execution provides many others, but we will not consider those here, we will only consider those exceptions interesting to those working in real mode[4].

Although exception handlers are user defined, the 80x86 hardware defines the exceptions that can occur. The 80x86 also assigns a fixed interrupt number to each of the exceptions. The following sections describe each of these exceptions in detail.

In general, an exception handler should preserve all registers. However, there are several special cases where you may want to tweak a register value before returning. For example, if you get a bounds violation, you may want to modify the value in the register specified by the bound instruction before returning. Nevertheless, you should not arbitrarily modify registers in an exception handling routine unless you intend to immediately abort the execution of your program.

17.3.1 Divide Error Exception (INT 0)

This exception occurs whenever you attempt to divide a value by zero or the quotient does not fit in the destination register when using the div or idiv instructions. Note that the FPU's fdiv and fdivr instructions do *not* raise this exception.

MS-DOS provides a generic divide exception handler that prints a message like "divide error" and returns control to MS-DOS. If you want to handle division errors yourself, you must write your own exception handler and patch the address of this routine into location 0:0.

On 8086, 8088, 80186, and 80188 processors, the return address on the stack points at the next instruction after the divide instruction. On the 80286 and later processors, the return address points at the beginning of the divide instruction (include any prefix bytes that appear). When a divide exception occurs, the 80x86 registers are unmodified; that is, they contain the values they held when the 80x86 first executed the div or idiv instruction.

When a divide exception occurs, there are three reasonable things you can attempt: abort the program (the easy way out), jump to a section of code that attempts to continue program execution in view of the error (e.g., as the user to reenter a value), or attempt to figure out why the error occurred, correct it, and reexecute the division instruction. Few people choose this last alternative because it is so difficult.

17.3.2 Single Step (Trace) Exception (INT 1)

The single step exception occurs after every instruction if the *trace* bit in the flags register is equal to one. Debuggers and other programs will often set this flag so they can trace the execution of a program.

When this exception occurs, the return address on the stack is the address of the *next* instruction to execute. The trap handler can decode this opcode and decide how to proceed. Most debuggers use the trace exception to check for *watchpoints* and other events that change dynamically during program execution. Debuggers that use the trace excep-

4. For more details on exceptions in protected mode, see the bibliography.

tion for single stepping often *disassemble* the next instruction using the return address on the stack as a pointer to that instruction's opcode bytes.

Generally, a single step exception handler should preserve *all* 80x86 registers and other state information. However, you will see an interesting use of the trace exception later in this text where we will purposely modify register values to make one instruction behave like another (see "The PC Keyboard" on page 1153).

Interrupt one is also shared by the debugging exceptions capabilities of 80386 and later processors. These processors provide on-chip support via *debugging registers*. If some condition occurs that matches a value in one of the debugging registers, the 80386 and later CPUs will generate a debugging exception that uses interrupt vector one.

17.3.3 Breakpoint Exception (INT 3)

The breakpoint exception is actually a trap, not an exception. It occurs when the CPU executes an int 3 instruction. However, we will consider it an exception since programmers rarely put int 3 instructions directly into their programs. Instead, a debugger like Codeview often manages the placement and removal of int 3 instructions.

When the 80x86 calls a breakpoint exception handling routine, the return address on the stack is the address of the next instruction after the breakpoint opcode. Note, however, that there are actually *two* int instructions that transfer control through this vector. Generally, though, it is the one-byte int 3 instruction whose opcode is 0cch; otherwise it is the two byte equivalent: 0cdh, 03h.

17.3.4 Overflow Exception (INT 4/INTO)

The overflow exception, like int 3, is technically a trap. The CPU only raises this exception when you execute an into instruction and the overflow flag is set. If the overflow flag is clear, the into instruction is effectively a nop, if the overflow flag is set, into behaves like an int 4 instruction. Programmers can insert an into instruction after an integer computation to check for an arithmetic overflow. Using into is equivalent to the following code sequence:

```
        « Some integer arithmetic code »
                jno     GoodCode
                int     4
GoodCode:
```

One big advantage to the into instruction is that it does not flush the pipeline or prefetch queue if the overflow flag is not set. Therefore, using the into instruction is a good technique if you provide a single overflow handler (that is, you don't have some special code for each sequence where an overflow could occur).

The return address on the stack is the address of the next instruction after into. Generally, an overflow handler does not return to that address. Instead, it will usually abort the program or pop the return address and flags off the stack and attempt the computation in a different way.

17.3.5 Bounds Exception (INT 5/BOUND)

Like into, the bound instruction (see "The INT, INTO, BOUND, and IRET Instructions" on page 292) will cause a conditional exception. If the specified register is outside the specified bounds, the bound instruction is equivalent to an int 5 instruction; if the register is within the specified bounds, the bound instruction is effectively a nop.

The return address that bound pushes is the address of the bound instruction itself, not the instruction following bound. If you return from the exception without modifying the

value in the register (or adjusting the bounds), you will generate an infinite loop because the code will reexecute the bound instruction and repeat this process over and over again.

One sneaky trick with the bound instruction is to generate a global minimum and maximum for an array of signed integers. The following code demonstrates how you can do this:

```
; This program demonstrates how to compute the minimum and maximum values
; for an array of signed integers using the bound instruction

                .xlist
                .286
                include     stdlib.a
                includelib  stdlib.lib
                .list

dseg            segment  para public 'data'

; The following two values contain the bounds for the BOUND instruction.

LowerBound      word     ?
UpperBound      word     ?

; Save the INT 5 address here:

OldInt5         dword    ?

; Here is the array we want to compute the minimum and maximum for:

Array           word     1, 2, -5, 345, -26, 23, 200, 35, -100, 20, 45
                word     62, -30, -1, 21, 85, 400, -265, 3, 74, 24, -2
                word     1024, -7, 1000, 100, -1000, 29, 78, -87, 60
ArraySize       =        ($-Array)/2

dseg            ends


cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg


; Our interrupt 5 ISR. It compares the value in AX with the upper and
; lower bounds and stores AX in one of them (we know AX is out of range
; by virtue of the fact that we are in this ISR).
;
; Note: in this particular case, we know that DS points at dseg, so this
; ISR will get cheap and not bother reloading it.
;
; Warning: This code does not handle the conflict between bound/int5 and
; the print screen key. Pressing prtsc while executing this code may
; produce incorrect results (see the text).

BoundISR        proc     near
                cmp      ax, LowerBound
                jl       NewLower

; Must be an upper bound violation.

                mov      UpperBound, ax
                iret

NewLower:       mov      LowerBound, ax
                iret
BoundISR        endp

Main            proc
                mov      ax, dseg
                mov      ds, ax
                meminit
```

```
; Begin by patching in the address of our ISR into int 5's vector.

                mov     ax, 0
                mov     es, ax
                mov     ax, es:[5*4]
                mov     word ptr OldInt5, ax
                mov     ax, es:[5*4 + 2]
                mov     word ptr OldInt5+2, ax

                mov     word ptr es:[5*4], offset BoundISR
                mov     es:[5*4 + 2], cs


; Okay, process the array elements. Begin by initializing the upper
; and lower bounds values with the first element of the array.

                mov     ax, Array
                mov     LowerBound, ax
                mov     UpperBound, ax

; Now process each element of the array:

                mov     bx, 2                   ;Start with second element.
                mov     cx, ArraySize
GetMinMax:      mov     ax, Array[bx]
                bound   ax, LowerBound
                add     bx, 2                   ;Move on to next element.
                loop    GetMinMax               ;Repeat for each element.

                printf
                byte    "The minimum value is %d\n"
                byte    "The maximum value is %d\n",0
                dword   LowerBound, UpperBound

; Okay, restore the interrupt vector:

                mov     ax, 0
                mov     es, ax
                mov     ax, word ptr OldInt5
                mov     es:[5*4], ax
                mov     ax, word ptr OldInt5+2
                mov     es:[5*4+2], ax


Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

If the array is large and the values appearing in the array are relatively random, this code demonstrates a fast way to determine the minimum and maximum values in the array. The alternative, comparing each element against the upper and lower bounds and storing the value if outside the range, is generally a slower approach. True, if the bound instruction causes a trap, this is *much* slower than the compare and store method. However, it a large array with random values, the bounds violation will rarely occur. Most of the time the bound instruction will execute in 7-13 clock cycles and it will not flush the pipeline or the prefetch queue[5].

Warning: IBM, in their infinite wisdom, decided to use int 5 as the *print screen* operation. The default int 5 handler will dump the current contents of the screen to the printer. This has two implications for those who would like to use the bound instruction in their programs. First, if you do not install your own int 5 handler and you execute a bound instruction that generates a bound exception, you will cause the machine to print the contents of the screen. Second, if you press the PrtSc key with your int 5 handler installed, BIOS will invoke your handler. The former case is a programming error, but this latter case means you have to make your bounds exception handler a little smarter. It should look at the byte pointed at by the return address. If this is an int 5 instruction opcode (0cdh), then you need to call the original int 5 handler, or simply return from interrupt (do you want them pressing the PrtSc key at that point?). If it is not an int 5 opcode, then this exception was probably raised by the bound instruction. Note that when executing a bound instruction the return address may not be pointing directly at a bound opcode (0c2h). It may be pointing at a prefix byte to the bound instruction (e.g., segment, addressing mode, or size override). Therefore, it is best to check for the int 5 opcode.

17.3.6 Invalid Opcode Exception (INT 6)

The 80286 and later processors raise this exception if you attempt to execute an opcode that does not correspond to a legal 80x86 instruction. These processors also raise this exception if you attempt to execute a bound, lds, les, lidt, or other instruction that requires a memory operand but you specify a register operand in the mod/rm field of the mod/reg/rm byte.

The return address on the stack points at the illegal opcode. By examining this opcode, you can extend the instruction set of the 80x86. For example, you could run 80486 code on an 80386 processor by providing subroutines that mimic the extra 80486 instructions (like bswap, cmpxchg, etc.).

17.3.7 Coprocessor Not Available (INT 7)

The 80286 and later processors raise this exception if you attempt to execute an FPU (or other coprocessor) instruction without having the coprocessor installed. You can use this exception to simulate the coprocessor in software.

On entry to the exception handler, the return address points at the coprocessor opcode that generated the exception.

17.4 Hardware Interrupts

Hardware interrupts are the form most engineers (as opposed to PC programmers) associate with the term *interrupt*. We will adopt this same strategy henceforth and will use the non-modified term “interrupt” to mean a hardware interrupt.

On the PC, interrupts come from many different sources. The primary sources of interrupts, however, are the PCs timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices. These devices connect to an Intel 8259A programmable interrupt controller (PIC) that prioritizes the interrupts and interfaces with the 80x86 CPU. The 8259A chip adds considerable complexity to the software that processes interrupts, so it makes perfect sense to discuss the PIC first, before trying to describe how the interrupt service routines have to deal with it. Afterwards, this section will briefly describe each device and the conditions under which

5. Note that on the 80486 and later processors, the bound instruction may actually be slower than the corresponding straight line code.

it interrupts the CPU. This text will fully describe many of these devices in later chapters, so this chapter will not go into a lot of detail except when discussing the timer interrupt.

17.4.1 The 8259A Programmable Interrupt Controller (PIC)

The 8259A (8259[6] or PIC, hereafter) programmable interrupt controller chip accepts interrupts from up to eight different devices. If any one of the devices requests service, the 8259 will toggle an interrupt output line (connected to the CPU) and pass a programmable interrupt vector to the CPU. You can *cascade* the device to support up to 64 devices by connecting nine 8259s together: eight of the devices with eight inputs each whose outputs become the eight inputs of the ninth device. A typical PC uses two of these devices to provide 15 interrupt inputs (seven on the *master* PIC with the eight input coming from the *slave* PIC to process its eight inputs)[7]. The sections following this one will describe the devices connected to each of those inputs, for now we will concentrate on what the 8259 does with those inputs. Nevertheless, for the sake of discussion, the following table lists the interrupt sources on the PC:

Table 66: 8259 Programmable Interrupt Controller Inputs

Input on 8259	80x86 INT	Device
IRQ 0	8	Timer chip
IRQ 1	9	Keyboard
IRQ 2	0Ah	Cascade for controller 2 (IRQ 8-15)
IRQ 3	0Bh	Serial port 2
IRQ 4	0Ch	Serial port 1
IRQ 5	0Dh	Parallel port 2 in AT, reserved in PS/2 systems
IRQ 6	0Eh	Diskette drive
IRQ 7	0Fh	Parallel port 1
IRQ 8/0	70h	Real-time clock
IRQ 9/1	71h	CGA vertical retrace (and other IRQ 2 devices)
IRQ 10/2	72h	Reserved
IRQ 11/3	73h	Reserved
IRQ 12/4	74h	Reserved in AT, auxiliary device on PS/2 systems
IRQ 13/5	75h	FPU interrupt
IRQ 14/6	76h	Hard disk controller
IRQ 15/7	77h	Reserved

The 8259 PIC is a very complex chip to program. Fortunately, all of the hard stuff has already been done for you by the BIOS when the system boots. We will not discuss how to initialize the 8259 in this text because that information is only useful to those writing operating systems like Linux, Windows, or OS/2. If you want your interrupt service routines to run correctly under DOS or any other OS, you must not reinitialize the PIC.

The PICs interface to the system through four I/O locations: ports 20h/0A0h and 21h/0A1h. The first address in each pair is the address of the master PIC (IRQ 0-7), the

6. The original 8259 was designed for Intel's 8080 system. The 8259A provided support for the 80x86 and some other features. Since almost no one uses 8259 chips anymore, this text will use the generic term 8259.
7. The original IBM PC and PC/XT machines only supported eight interrupts via one 8259 chip. IBM, and virtually all clone manufacturers, added the second PIC in PC/AT and later designs.

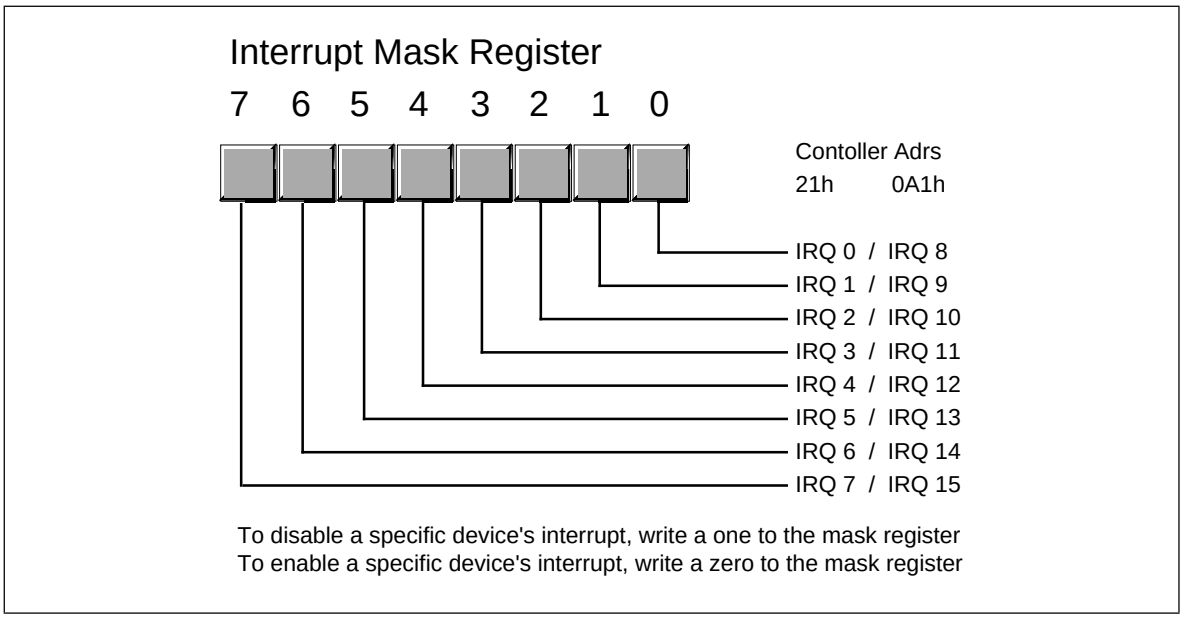


Figure 17.1 8259 Interrupt Mask Register

second address in each pair corresponds to the slave PIC (IRQ 8-15). Port 20h/0A0h is a read/write location to which you write PIC commands and read PIC status, we will refer to this as the *command register* or the *status register*. The command register is write only, the status register is read only. They just happen to share the same I/O location. The read/write lines on the PIC determine which register the CPU accesses. Port 21h/0A1h is a read/write location that contains the interrupt mask register, we will refer to this as the *mask register*. Choose the appropriate address depending upon which interrupt controller you want to use.

The interrupt mask register is an eight bit register that lets you individually enable and disable interrupts from devices on the system. This is similar to the actions of the cli and sti instructions, but on a device by device basis. Writing a zero to the corresponding bit *enables* that device's interrupts. Writing a one *disables* interrupts from the affected device. Note that this is non-intuitive. Figure 17.1 provides the layout of the interrupt mask register.

When changing bits in the mask register, it is important that you not simply load al with a value and output it directly to the mask register port. Instead, you should read the mask register and then logically or in or and out the bits you want to change; finally, you can write the output back to the mask register. The following code sequence enables COM1: interrupts without affecting any others:

```
in      al, 21h             ;Read existing bits.
and     al, 0efh            ;Turn on IRQ 4 (COM1).
out     21h, al             ;Write result back to PIC.
```

The command register provides lots of options, but there are only three commands you would want to execute on this chip that are compatible with the BIOS' initialization of the 8259: sending an end of interrupt command and sending one of two read status register commands.

One a specific interrupt occurs, the 8259 masks all further interrupts from that device until is receives an *end of interrupt* signal from the interrupt service routine. On PCs running DOS, you accomplish this by writing the value 20h to the command register. The following code does this:

```
mov     al, 20h
out     20h, al             ;Port 0A0h if IRQ 8-15.
```

You must send exactly one end of interrupt command to the PIC for each interrupt you service. If you do not send the end of interrupt command, the PIC will not honor any more interrupts from that device; if you send two or more end of interrupt commands, there is the possibility that you will accidentally acknowledge a new interrupt that may be pending and you will lose that interrupt.

For some interrupt service routines you write, your ISR will not be the only ISR that an interrupt invokes. For example, the PC's BIOS provides an ISR for the timer interrupt that maintains the time of day. If you patch into the timer interrupt, you will need to call the PC BIOS' timer ISR so the system can properly maintain the time of day and handle other timing related chores (see "Chaining Interrupt Service Routines" on page 1010). However, the BIOS' timer ISR outputs the end of interrupt command. Therefore, you should not output the end of interrupt command yourself, otherwise the BIOS will output a second end of interrupt command and you may lose an interrupt in the process.

The other two commands you can send the 8259 let you select whether to read the *in-service register* (ISR) or the *interrupt request register* (IRR). The in-service register contains set bits for each active ISR (because the 8259 allows prioritized interrupts, it is quite possible that one ISR has been interrupted by a higher priority ISR). The interrupt request register contains set bits in corresponding positions for interrupts that have not yet been serviced (probably because they are a lower priority interrupt than the interrupt currently being serviced by the system). To read the in-service register, you would execute the following statements:

```
; Read the in-service register in PIC #1 (at I/O address 20h)

                mov     al, 0bh
                out     20h, al
                in      al, 20h
```

To read the interrupt request register, you would use the following code:

```
; Read the interrupt request register in PIC #1 (at I/O address 20h)

                mov     al, 0ah
                out     20h, al
                in      al, 20h
```

Writing any other values to the command port may cause your system to malfunction.

17.4.2 The Timer Interrupt (INT 8)

The PC's motherboard contains an 8254 compatible timer chip. This chip contains three timer channels, one of which generates interrupts every 55 msec (approximately). This is about once every $^{1}/_{18.2}$ seconds. You will often hear this interrupt referred to as the "eighteenth second clock." We will simply call it the timer interrupt.

The timer interrupt vector is probably the most commonly patched interrupt in the system. It turns out there are *two* timer interrupt vectors in the system. Int 8 is the hardware vector associated with the timer interrupt (since it comes in on IRQ 0 on the PIC). Generally, you should *not* patch this interrupt if you want to write a timer ISR. Instead, you should patch the second timer interrupt, interrupt 1ch. The BIOS' timer interrupt handler (int 8) executes an `int 1ch` instruction before it returns. This gives a user patched routine access to the timer interrupt. Unless you are willing to duplicate the BIOS and DOS timer code, you should never completely replace the existing timer ISR with one of your own, you should always ensure that the BIOS and DOS ISRs execute in addition to your ISR. Patching into the int 1ch vector is the easiest way to do this.

Even replacing the int 1ch vector with a pointer to your ISR is very dangerous. The timer interrupt service routine is the one most commonly patched by various resident programs (see "Resident Programs" on page 1025). By simply writing the address of your ISR into the timer interrupt vector, you may disable such resident programs and cause your

system to malfunction. To solve this problem, you need to create an *interrupt chain*. For more details, see the section "Chaining Interrupt Service Routines" on page 1010.

By default the timer interrupt is always enabled on the interrupt controller chip. Indeed, disabling this interrupt may cause your system to crash or otherwise malfunction. At the very least, you system will not maintain the correct time if you disable the timer interrupt.

17.4.3 The Keyboard Interrupt (INT 9)

The keyboard microcontroller on the PC's motherboard generates *two* interrupts on each keystroke – one when you press a key and one when you release it. This is on IRQ 1 on the master PIC. The BIOS responds to this interrupt by reading the keyboard's *scan code*, converting this to an ASCII character, and storing the scan and ASCII codes away in the system *type ahead buffer*.

By default, this interrupt is always enabled. If you disable this interrupt, the system will not be able to respond to any keystrokes, including ctrl-alt-del. Therefore, your programs should always reenable this interrupt if they ever disable it.

For more information on the keyboard interrupt, see "The PC Keyboard" on page 1153.

17.4.4 The Serial Port Interrupts (INT 0Bh and INT 0Ch)

The PC uses two interrupts, IRQ 3 and IRQ 4, to support interrupt driven serial communications. The 8250 (or compatible) serial communications controller chip (SCC) generates an interrupt in one of four situations: a character arriving over the serial line, the SCC finishes the transmission of a character and is requesting another, an error occurs, or a status change occurs. The SCC activates the same interrupt line (IRQ 3 or 4) for all four interrupt sources. The interrupt service routine is responsible for determining the exact nature of the interrupt by interrogating the SCC.

By default, the system disables IRQ 3 and IRQ 4. If you install a serial ISR, you will need to clear the interrupt mask bit in the 8259 PIC before it will respond to interrupts from the SCC. Furthermore, the SCC design includes its own interrupt mask. You will need to enable the interrupt masks on the SCC chip as well. For more information on the SCC, see "The PC Serial Ports" on page 1223.

17.4.5 The Parallel Port Interrupts (INT 0Dh and INT 0Fh)

The parallel port interrupts are an enigma. IBM designed the original system to allow two parallel port interrupts and then promptly designed a printer interface card that didn't support the use of interrupts. As a result, almost no DOS based software today uses the parallel port interrupts (IRQ 5 and IRQ 7). Indeed, on the PS/2 systems IBM reserved IRQ5 which they formerly used for LPT2:.

However, these interrupts have not gone to waste. Many devices which IBM's engineers couldn't even conceive when designing the first PC have made good use of these interrupts. Examples include SCSI cards and sound cards. Many devices today include "interrupt jumpers" that let you select IRQ 5 or IRQ 7 when installing the device.

Since IRQ 5 and IRQ 7 find such little use as parallel port interrupts, we will effectively ignore the "parallel port interrupts" in this text.

17.4.6 The Diskette and Hard Drive Interrupts (INT 0Eh and INT 76h)

The floppy and hard disk drives generate interrupts at the completion of a disk operation. This is a very useful feature for multitasking systems like OS/2, Linux, or Windows. While the disk is reading or writing data, the CPU can go execute instructions for another process. When the disk finishes the read or write operation, it interrupts the CPU so it can resume the original task.

While managing the disk drives would be an interesting topic to cover in this text, this book is already long enough. Therefore, this text will avoid discussing the disk drive interrupts (IRQ 6 and IRQ 14) in the interest of saving some space. There are many texts that cover low level disk I/O in assembly language, see the bibliography for details.

By default, the floppy and hard disk interrupts are always enabled. You should not change this status if you intend to use the disk drives on your system.

17.4.7 The Real-Time Clock Interrupt (INT 70h)

PC/AT and later machines included a CMOS real-time clock. This device is capable of generating timer interrupts in multiples of 976 µsec (let's call it 1 msec). By default, the real-time clock interrupt is disabled. You should only enable this interrupt if you have an int 70h ISR installed.

17.4.8 The FPU Interrupt (INT 75h)

The 80x87 FPU generates an interrupt whenever a floating point exception occurs. On CPUs with built-in FPUs (80486DX and better) there is a bit in one of the control register you can set to simulate a vectored interrupt. BIOS generally initializes such bits for compatibility with existing systems.

By default, BIOS disables the FPU interrupt. Most programs that use the FPU explicitly test the FPU's status register to determine if an error occurs. If you want to allow FPU interrupts, you must enable the interrupts on the 8259 *and* on the 80x87 FPU.

17.4.9 Nonmaskable Interrupts (INT 2)

The 80x86 chips actually provide *two* interrupt input pins. The first is the *maskable* interrupt. This is the pin to which the 8259 PIC connects. This interrupt is maskable because you can enable or disable it with the cli and sti instructions. The *nonmaskable* interrupt, as its name implies, cannot be disabled under software control. Generally, PCs use this interrupt to signal a memory parity error, although certain systems use this interrupt for other purposes as well. Many older PC systems connect the FPU to this interrupt.

This interrupt cannot be masked, so it is always enabled by default.

17.4.10 Other Interrupts

As mentioned in the section on the 8259 PIC, there are several interrupts reserved by IBM. Many systems use the reserved interrupts for the mouse or for other purposes. Since such interrupts are inherently system dependent, we will not describe them here.

17.5 Chaining Interrupt Service Routines

Interrupt service routines come in two basic varieties – those that need exclusive access to an interrupt vector and those that must share an interrupt vector with several other ISRs. Those in the first category include error handling ISRs (e.g., divide error or overflow) and certain device drivers. The serial port is a good example of a device that rarely has more than one ISR associated with it at any one given time[8]. The timer, real-time clock, and keyboard ISRs generally fall into the latter category. It is not at all uncommon to find several ISRs in memory sharing each of these interrupts.

Sharing an interrupt vector is rather easy. All an ISR needs to do to share an interrupt vector is to save the old interrupt vector when installing the ISR (something you need to do anyway, so you can restore the interrupt vector when your code terminates) and then call the original ISR before or after you do your own ISR processing. If you've saved away the address of the original ISR in the dseg double word variable OldIntVect, you can call the original ISR with the following code:

```
; Presumably, DS points at DSEG at this point.

                pushf                   ;Simulate an INT instruction by pushing
                call    OldIntVect      ; the flags and making a far call.
```

Since OldIntVect is a dword variable, this code generates a far call to the routine whose segmented address appears in the OldIntVect variable. This code does *not* jump to the location of the OldIntVect variable.

Many interrupt service routines do not modify the ds register to point at a local data segment. In fact, some simple ISRs do not change any of the segment registers. In such cases it is common to put any necessary variables (especially the old segment value) directly in the code segment. If you do this, your code could *jump* directly to the original ISR rather than calling it. To do so, you would just use the code:

```
MyISR           proc    near
                  .
                  .
                  .
                jmp     cs:OldIntVect
MyISR           endp

OldIntVect      dword   ?
```

This code sequence passes along your ISR's flags and return address as the flag and return address values to the original ISR. This is fine, when the original ISR executes the iret instruction, it will return directly to the interrupted code (assuming it doesn't pass control to some other ISR in the chain).

The OldIntVect variable *must* be in the code segment if you use this technique to transfer control to the original ISR. After all, when you executing the jmp instruction above, you must have already restored the state of the CPU, including the ds register. Therefore, you have no idea what segment ds is pointing at, and it probably isn't pointing at your local data segment. Indeed, the only segment register whose value is known to you is cs, so you must keep the vector address in your code segment.

The following simple program demonstrates interrupt chaining. This short program patches into the int 1ch vector. The ISR counts off seconds and notifies the main program as each second passes. The main program prints a short message every second. When 10 seconds have expired, this program removes the ISR from the interrupt chain and terminates.

```
; TIMER.ASM
; This program demonstrates how to patch into the int 1Ch timer interrupt
; vector and create an interrupt chain.
```

8. There is no reason this has to be this way, it's just that most people rarely run two programs at the same time which must both be accessing the serial port.

```
                .xlist
                .286
                include     stdlib.a
                includelib  stdlib.lib
                .list

dseg            segment  para public 'data'

; The TIMERISR will update the following two variables.
; It will update the MSEC variable every 55 ms.
; It will update the TIMER variable every second.

MSEC            word     0
TIMER           word     0

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; The OldInt1C variable must be in the code segment because of the
; way TimerISR transfers control to the next ISR in the int 1Ch chain.

OldInt1C        dword    ?


; The timer interrupt service routine.
; This guy increment MSEC variable by 55 on every interrupt.
; Since this interrupt gets called every 55 msec (approx) the
; MSEC variable contains the current number of milliseconds.
; When this value exceeds 1000 (one second), the ISR subtracts
; 1000 from the MSEC variable and increments TIMER by one.

TimerISR        proc     near
                push     ds
                push     ax
                mov      ax, dseg
                mov      ds, ax

                mov      ax, MSEC
                add      ax, 55        ;Interrupt every 55 msec.
                cmp      ax, 1000
                jb       SetMSEC
                inc      Timer         ;A second just passed.
                sub      ax, 1000      ;Adjust MSEC value.
SetMSEC:        mov      MSEC, ax
                pop      ax
                pop      ds
                jmp      cseg:OldInt1C ;Transfer to original ISR.
TimerISR        endp

Main            proc
                mov      ax, dseg
                mov      ds, ax
                meminit

; Begin by patching in the address of our ISR into int 1ch's vector.
; Note that we must turn off the interrupts while actually patching
; the interrupt vector and we must ensure that interrupts are turned
; back on afterwards; hence the cli and sti instructions. These are
; required because a timer interrupt could come along between the two
; instructions that write to the int 1Ch interrupt vector. This would
; be a big mess.

                mov      ax, 0
                mov      es, ax
                mov      ax, es:[1ch*4]
                mov      word ptr OldInt1C, ax
                mov      ax, es:[1ch*4 + 2]
```

```
                mov     word ptr OldInt1C+2, ax

                cli
                mov     word ptr es:[1Ch*4], offset TimerISR
                mov     es:[1Ch*4 + 2], cs
                sti

; Okay, the ISR updates the TIMER variable every second.
; Continuously print this value until ten seconds have
; elapsed. Then quit.

                mov     Timer, 0
TimerLoop:      printf
                byte    "Timer = %d\n",0
                dword   Timer
                cmp     Timer, 10
                jbe     TimerLoop

; Okay, restore the interrupt vector. We need the interrupts off
; here for the same reason as above.

                mov     ax, 0
                mov     es, ax
                cli
                mov     ax, word ptr OldInt1C
                mov     es:[1Ch*4], ax
                mov     ax, word ptr OldInt1C+2
                mov     es:[1Ch*4+2], ax
                sti

Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

17.6 Reentrancy Problems

A minor problem develops with developing ISRs, what happens if you enable interrupts while in an ISR and a second interrupt from the same device comes along? This would interrupt the ISR and then *reenter* the ISR from the beginning. Many applications do not behave properly under these conditions. An application that can properly handle this situation is said to be *reentrant*. Code segments that do not operate properly when reentered are *nonreentrant*.

Consider the TIMER.ASM program in the previous section. This is an example of a nonreentrant program. Suppose that while executing the ISR, it is interrupted at the following point:

```
TimerISR        proc    near
                push    ds
                push    ax
                mov     ax, dseg
                mov     ds, ax

                mov     ax, MSEC
                add     ax, 55      ;Interrupt every 55 msec.
                cmp     ax, 1000
                jb      SetMSEC
```

```
; <<<<< Suppose the interrupt occurs at this point >>>>>

                inc     Timer           ;A second just passed.
                sub     ax, 1000        ;Adjust MSEC value.
SetMSEC:        mov     MSEC, ax
                pop     ax
                pop     ds
                jmp     cseg:OldInt1C ;Transfer to original ISR.
TimerISR        endp
```

Suppose that, on the first invocation of the interrupt, MSEC contains 950 and Timer contains three. If a second interrupt occurs and the specified point above, ax will contain 1005. So the interrupt suspends the ISR and reenters it from the beginning. Note that TimerISR is nice enough to preserve the ax register containing the value 1005. When the second invocation of TimerISR executes, it finds that MSEC still contains 950 because the first invocation has yet to update MSEC. Therefore, it adds 55 to this value, determines that it exceeds 1000, increments Timer (it becomes four) and then stores five into MSEC. Then it returns (by jumping to the next ISR in the int 1ch chain). Eventually, control returns the first invocation of the TimerISR routine. At this time (less than 55 msec after updating Timer by the second invocation) the TimerISR code increments the Timer variable again and updates MSEC to five. The problem with this sequence is that it has incremented the Timer variable twice in less than 55 msec.

Now you might argue that hardware interrupts always clear the interrupt disable flag so it would not be possible for this interrupt to be reentered. Furthermore, you might argue that this routine is so short, it would never take more than 55 msec to get to the noted point in the code above. However, you are forgetting something: some other timer ISR could be in the system that calls *your* code after it is done. That code could take 55 msec and just happen to turn the interrupts back on, making it perfectly possible that your code could be reentered.

The code between the mov ax, MSEC and mov MSEC, ax instructions above is called a *critical region* or *critical section*. A program must not be reentered while it is executing in a critical region. Note that having critical regions does not mean that a program is not reentrant. Most programs, even those that are reentrant, have various critical regions. The key is to prevent an interrupt that could cause a critical region to be reentered while in that critical region. The easiest way to prevent such an occurrence is to *turn off the interrupts* while executing code in a critical section. We can easily modify the TimerISR to do this with the following code:

```
TimerISR        proc    near
                push    ds
                push    ax
                mov     ax, dseg
                mov     ds, ax

; Beginning of critical section, turn off interrupts.

                pushf                   ;Preserve current I flag state.
                cli                     ;Make sure interrupts are off.

                mov     ax, MSEC
                add     ax, 55          ;Interrupt every 55 msec.
                cmp     ax, 1000
                jb      SetMSEC

                inc     Timer           ;A second just passed.
                sub     ax, 1000        ;Adjust MSEC value.
SetMSEC:        mov     MSEC, ax

; End of critical region, restore the I flag to its former glory.

                popf
```

```
                pop     ax
                pop     ds
                jmp     cseg:OldInt1C;Transfer to original ISR.
TimerISR        endp
```

We will return to the problem of reentrancy and critical regions in the next two chapters of this text.

17.7 The Efficiency of an Interrupt Driven System

Interrupts introduce a considerable amount of complexity to a software system (see "Debugging ISRs" on page 1020). One might ask if using interrupts is really worth the trouble. The answer of course, is yes. Why else would people use interrupts if they were proven not to be worthwhile? However, interrupts are like many other nifty things in computer science – they have their place; if you attempt to use interrupts in an inappropriate fashion they will only make things worse for you.

The following sections explore the efficiency aspects of using interrupts. As you will soon discover, an interrupt driven system is usually superior despite the complexity. However, this is not always the case. For many systems, alternative methods provide better performance.

17.7.1 Interrupt Driven I/O vs. Polling

The whole purpose of an interrupt driven system is to allow the CPU to continue processing instructions while some I/O activity occurs. This is in direct contrast to a *polling system* where the CPU continually tests an I/O device to see if the I/O operation is complete. In an interrupt driven system, the CPU goes about its business and the I/O device interrupts it when it needs servicing. This is generally much more efficient than wasting CPU cycles polling a device while it is not ready.

The serial port is a perfect example of a device that works extremely well with interrupt driven I/O. You can start a communication program that begins downloading a file over a modem. Each time a character arrives, it generates an interrupt and the communication program starts up, buffers the character, and then returns from the interrupt. In the meantime, another program (like a word processor) can be running with almost no performance degradation since it takes so little time to process the serial port interrupts.

Contrast the above scenario with one where the serial communication program continually polls the serial communication chip to see if a character has arrived. In this case the CPU spends all of its time looking for an input character even though one rarely (in CPU terms) arrives. Therefore, no CPU cycles are left over to do other processing like running your word processor.

Suppose interrupts were not available and you wanted to allow background downloads while using your word processing program. Your word processing program would have to test the input data on the serial port once every few milliseconds to keep from losing any data. Can you imagine how difficult such a word processor would be to write? An interrupt system is the clear choice in this case.

If downloading data while word processing seems far fetched, consider a more simple case – the PC's keyboard. Whenever a keypress interrupt occurs, the keyboard ISR reads the key pressed and saves it in the system type ahead buffer for the moment when the application wants to read the keyboard data. Can you imagine how difficult it would be to write applications if you had to constantly poll the keyboard port yourself to keep from losing characters? Even in the middle of a long calculation? Once again, interrupts provide an easy solution.

17.7.2 Interrupt Service Time

Of course, the serial communication system just described is an example of a *best case scenario*. The communication program takes so little time to do its job that most of the time is left over for the word processing program. However, were to you run a different interrupt driven I/O system, for example, copying files from one disk to another, the interrupt service routine would have a noticeable impact on the performance of the word processing system.

Two factors control an ISR's impact on a computer system: the *frequency of interrupts* and the *interrupt service time*. The frequency is how many times per second (or other time measurement) a particular interrupt occurs. The interrupt service time is how long the ISR takes to service the interrupt.

The nature of the frequency varies according to source of the interrupt. For example, the timer chip generates evenly spaced interrupts about 18 times per second, likewise, a serial port receiving at 9600bps generates better than 100 interrupts per second. On the other hand, the keyboard rarely generates more than about 20 interrupts per second and they are not very regular.

The interrupt service time is obviously dependent upon the number of instructions the ISR must execute. The interrupt service time is also dependent upon the particular CPU and clock frequency. The same ISR executing identical instructions on two CPUs will run in less time on a faster machine.

The amount of time an interrupt service routine takes to handle an interrupt, multiplied by the frequency of the interrupt, determines the impact the interrupt will have on system performance. Remember, every CPU cycle spent in an ISR is one less cycle available for your application programs. Consider the timer interrupt. Suppose the timer ISR takes 100 µsec to complete its tasks. This means that the timer interrupt consumes 1.8 msec out of every second, or about 0.18% of the total computer time. Using a faster CPU will reduce this percentage (by reducing the time spent in the ISR); using a slower CPU will increase the percentage. Nevertheless, you can see that a short ISR such as this one will not have a significant effect on overall system performance.

One hundred microseconds is fast for a typical timer ISR, especially when your system has several timer ISRs chained together. However, even if the timer ISR took ten times as long to execute, it would only rob the system of less than 2% of the available CPU cycles. Even if it took 100 times longer (10 msec), there would only be an 18% performance degradation; most people would barely notice such a degradation[9].

Of course, one cannot allow the ISR to take as much time as it wants. Since the timer interrupt occurs every 55 msec, the *maximum* time the ISR can use is just under 55msec. If the ISR requires more time than there is between interrupts, the system will eventually lose an interrupt. Furthermore, the system will spend all its time servicing the interrupt rather than accomplishing anything else.

For many systems, having an ISR that consumes as much as 10% of the overall CPU cycles will not prove to a problem. However, before you go off and start designing slow interrupt service routines, you should remember that your ISR is probably not the only ISR in the system. While your ISR is consuming 25% of the CPU cycles, there may be another ISR that is doing the same thing; and another, and another, and… Furthermore, there may be some ISRs that require fast servicing. For example, a serial port ISR may need to read a character from the serial communications chip each millisecond or so. If your timer ISR requires 4 msec to execute and does so with the interrupts turned off, the serial port ISR will miss some characters.

Ultimately, of course, you would like to write ISRs so they are as fast as possible so they have as little impact on system performance as they can. This is one of the main rea-

9. As a general rule, people begin to notice a real difference in performance between 25 and 50%. It isn't instantly obvious until about 50% (i.e., running at one-half the speed).

sons most ISRs for DOS are still written in assembly language. Unless you are designing an *embedded system*, one in which the PC runs only your application, you need to realize that your ISRs must coexist with other ISRs and applications; you do not want the performance of your ISR to adversely affect the performance of other code in the system.

17.7.3 Interrupt Latency

Interrupt latency is the time between the point a device signals that it needs service and the point where the ISR provides the needed service. This is not instantaneous! At the very least, the 8259 PIC needs to signal the CPU, the CPU needs to interrupt the current program, push the flags and return address, obtain the ISR address, and transfer control to the ISR. The ISR may need to push various registers, set up certain variables, check device status to determine the source of the interrupt, and so on. Furthermore, there may be other ISRs chained into the interrupt vector before you and they execute to completion before transferring control to your ISR that actually services the device. Eventually, the ISR actually does whatever it is that the device needs done. In the best case on the fastest microprocessors with simple ISRs, the latency could be under a microsecond. On slower systems, with several ISRs in a chain, the latency could be as bad as several *milli*seconds.

For some devices, the interrupt latency is more important than the actual interrupt service time. For example, an input device may only interrupt the CPU once every 10 seconds. However, that device may be incapable of holding the data on its input port for more than a millisecond. In theory, any interrupt service time less than 10 seconds is fine; but the CPU must read the data within one millisecond of its arrival or the system will lose the data.

Low interrupt latency (that is, responding quickly) is very important in many applications. Indeed, in some applications the latency requirements are so strict that you have to use a very fast CPU or you have to abandon interrupts altogether and go back to polling. *What a minute!* Isn't polling less efficient than an interrupt driven system? How will polling improve things?

An interrupt driven I/O system improves system performance by allowing the CPU to work on other tasks in between I/O operations. In principle, servicing interrupts takes very little CPU time compared the arrival of interrupts to the system. By using interrupt driven I/O, you can use all those other CPU cycles for some other purpose. However, suppose the I/O device is producing service requests at such a rate that there are no free CPU cycles. Interrupt driven I/O will provide few benefits in this case.

For example, suppose we have an eight bit I/O device connected to two I/O ports. Suppose bit zero of port 310h contains a one if data is available and a zero otherwise. If data is available, the CPU must read the eight bits at port 311h. Reading port 311h clears bit zero of port 310h until the next byte arrives. If you wanted to read 8192 bytes from this port, you could do this with the following short segment of code:

```
                mov     cx, 8192
                mov     dx, 310h
                lea     bx, Array               ;Point bx at storage buffer
DataAvailLp:    in      al, dx                  ;Read status port.
                shr     al, 1                   ;Test bit zero.
                jnc     DataAvailLp             ;Wait until data is
available.
                inc     dx                      ;Point at data port.
                in      al, dx                  ;Read data.
                mov     [bx], al                ;Store data into buffer.
                inc     bx                      ;Move on to next array
element.
                dec     dx                      ;Point back at status port.
                loop    DataAvailLp             ;Repeat 8192 times.
                  .
                  .
                  .
```

This code uses a classical polling loop (DataAvailLp) to wait for each available character. Since there are only three instructions in the polling loop, this loop can probably execute in just under a microsecond[10]. So it might take as much as one microsecond to determine that data is available, in which case the code falls through and by the second instruction in the sequence we've read the data from the device. Let's be generous and say that takes another microsecond. Suppose, instead, we use a interrupt service routine. A *well-written* ISR combined with a good system hardware design will probably have latencies measured in microseconds.

To measure the *best case* latency we could hope to achieve would require some sort of hardware timer than begins counting once an interrupt event occurs. Upon entry into our interrupt service routine we could read this counter to determine how much time has passed between the interrupt and its service. Fortunately, just such a device exists on the PC – the 8254 timer chip that provides the source of the 55 msec interrupt.

The 8254 timer chip actually contains three separate timers: timer #0, timer #1, and timer #2. The first timer (timer #0) provides the clock interrupt, so it will be the focus of our discussion. The timer contains a 16 bit register that the 8254 decrements at regular intervals (1,193,180 times per second). Once the timer hits zero, it generates an interrupt on the 8259 IRQ 0 line and then wraps around to 0FFFFh and continues counting down from that point. Since the counter automatically resets to 0FFFFh after generating each interrupt, this means that the 8254 timer generates interrupts every 65,536 / 1,193,180 seconds, or once every 54.9254932198 msec, which is 18.2064819336 times per second. We'll just call these once every 55 msec or 18 (or 18.2) times per second, respectively. Another way to view this is that the 8254 decrements the counter once every 838 nanoseconds (or 0.838 µsec).

The following short assembly language program measures interrupt latency by patching into the int 8 vector. Whenever the timer chip counts down to zero, it generates an interrupt that directly calls this program's ISR. The ISR quickly reads the timer chip's counter register, negates the value (so 0FFFFh becomes one, 0FFFEh becomes two, etc.), and then adds it to a running total. The ISR also increments a counter so that it can keep track of the number of times it has added a counter value to the total. Then the ISR jumps to the original int 8 handler. The main program, in the mean time, simply computes and displays the current average read from the counter. When the user presses any key, this program terminates.

```
; This program measures the latency of an INT 08 ISR.
; It works by reading the timer chip immediately upon entering
; the INT 08 ISR By averaging this value for some number of
; executions, we can determine the average latency for this
; code.

                .xlist
                .386
                option      segment:use16
                include     stdlib.a
                includelib  stdlib.lib
                .list


cseg            segment   para public 'code'
                assume    cs:cseg, ds:nothing

; All the variables are in the code segment in order to reduce ISR
; latency (we don't have to push and set up DS, saving a few instructions
; at the beginning of the ISR).

OldInt8         dword     ?
SumLatency      dword     0
```

10. On a fast CPU (.e.g, 100 MHz Pentium), you might expect this loop to execute in much less time than one microsecond. However, the in instruction is probably going to be quite slow because of the wait states associated with external I/O devices.

```
Executions      dword   0
Average         dword   0

; This program reads the 8254 timer chip. This chip counts from
; 0FFFFh down to zero and then generates an interrupt. It wraps
; around from 0 to 0FFFFh and continues counting down once it
; generates the interrupt.
;
; 8254 Timer Chip port addresses:

Timer0_8254     equ     40h
Cntrl_8254      equ     43h


; The following ISR reads the 8254 timer chip, negates the result
; (because the timer counts backwards), adds the result to the
; SumLatency variable, and then increments the Executions variable
; that counts the number of times we execute this code. In the
; mean time, the main program is busy computing and displaying the
; average latency time for this ISR.
;
; To read the 16 bit 8254 counter value, this code needs to
; write a zero to the 8254 control port and then read the
; timer port twice (reads the L.O. then H.O. bytes). There
; needs to be a short delay between reading the two bytes
; from the same port address.

TimerISR        proc    near
                push    ax
                mov     eax, 0          ;Ch 0, latch & read data.
                out     Cntrl_8254, al  ;Output to 8253 cmd register.
                in      al, Timer0_8254 ;Read latch #0 (LSB) & ignore.
                mov     ah, al
                jmp     SettleDelay     ;Settling delay for 8254 chip.
SettleDelay:    in      al, Timer0_8254 ;Read latch #0 (MSB)
                xchg    ah, al
                neg     ax              ;Fix, 'cause timer counts down.
                add     cseg:SumLatency, eax
                inc     cseg:Executions
                pop     ax
                jmp     cseg:OldInt8
TimerISR        endp



Main            proc
                meminit

; Begin by patching in the address of our ISR into int 8's vector.
; Note that we must turn off the interrupts while actually patching
; the interrupt vector and we must ensure that interrupts are turned
; back on afterwards; hence the cli and sti instructions. These are
; required because a timer interrupt could come along between the two
; instructions that write to the int 8 interrupt vector. Since the
; interrupt vector is in an inconsistent state at that point, this
; could cause the system to crash.

                mov     ax, 0
                mov     es, ax
                mov     ax, es:[8*4]
                mov     word ptr OldInt8, ax
                mov     ax, es:[8*4 + 2]
                mov     word ptr OldInt8+2, ax

                cli
                mov     word ptr es:[8*4], offset TimerISR
                mov     es:[8*4 + 2], cs
                sti

; First, wait for the first call to the ISR above. Since we will be dividing
```

```
; by the value in the Executions variable, we need to make sure that it is
; greater than zero before we do anything.

Wait4Non0:      cmp     cseg:Executions, 0
                je      Wait4Non0


; Okay, start displaying the good values until the user presses a key at
; the keyboard to stop everything:

DisplayLp:      mov     eax, SumLatency
                cdq                             ;Extends eax->edx.
                div     Executions
                mov     Average, eax
                printf
                byte    "Count: %ld, average: %ld\n",0
                dword   Executions, Average

                mov     ah, 1                   ;Test for keystroke.
                int     16h
                je      DisplayLp
                mov     ah, 0                   ;Read that keystroke.
                int     16h



; Okay, restore the interrupt vector. We need the interrupts off
; here for the same reason as above.

                mov     ax, 0
                mov     es, ax
                cli
                mov     ax, word ptr OldInt8
                mov     es:[8*4], ax
                mov     ax, word ptr OldInt8+2
                mov     es:[8*4+2], ax
                sti

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             db       1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

On a 66 MHz 80486 DX/2 processor, the above code reports an average value of 44 after it has run for about 10,000 iterations. This works out to about 37 µsec between the device signalling the interrupt and the ISR being able to process it[11]. *The latency of polled I/O would probably be an order of magnitude less than this!*

Generally, if you have some high speed application like audio or video recording or playback, you probably cannot afford the latencies associated with interrupt I/O. On the other hand, such applications demand such high performance out of the system, that you probably wouldn't have any CPU cycles left over to do other processing while waiting for I/O.

11. Patching into the int 1Ch interrupt vector produces latencies in the 137 µsec range.

Another issue with respect to ISR latency is *latency consistency*. That is, is there the same amount of latency from interrupt to interrupt? Some ISRs can tolerate considerable latency as long as it is consistent (that is, the latency is roughly the same from interrupt to interrupt). For example, suppose you want to patch into the timer interrupt so you can read an input port every 55 msec and store this data away. Later, when processing the data, your code might work under the assumption that the data readings are 55 msec (or 54.9…) apart. This might not be true if there are other ISRs in the timer interrupt chain before your ISR. For example, there may be an ISR that counts off 18 interrupts and then executes some code sequence that requires 10 msec. This means that 16 out of every 18 interrupts your data collection routine would collect data at 55 msec intervals right on the nose. But when that 18th interrupt occurs, the other timer ISR will delay 10 msec before passing control to your routine. This means that your 17th reading will be 65 msec since the last reading. Don't forget, the timer chip is still counting down during all of this, that means there are now only 45 msec to the next interrupt. Therefore, your 18th reading would occur 45 msec after the 17th. Hardly a consistent pattern. If your ISR needs a consistent latencies, you should try to install your ISR as early in the interrupt chain as possible.

17.7.4 Prioritized Interrupts

Suppose you have the interrupts turned off for a brief spell (perhaps you are processing some interrupt) and *two* interrupt requests come in while the interrupts are off. What happens when you turn the interrupts back on? Which interrupt will the CPU first service? The obvious answer would be "whichever interrupt occurred first." However, suppose the both occurred at exactly the same time (or, at least, within a short enough time frame that we cannot determine which occurred first), or maybe, as is really the case, the 8259 PIC cannot keep track of which interrupt occurred first? Furthermore, what if one interrupt is more important that another? Suppose for example, that one interrupt tells that the user has just pressed a key on the keyboard and a second interrupt tells you that your nuclear reactor is about to melt down if you don't do something in the next 100 µsec. Would you want to process the keystroke first, even if its interrupt came in first? Probably not. Instead, you would want to *prioritizes* the interrupts on the basis of their importance; the nuclear reactor interrupt is probably a little more important than the keystroke interrupt, you should probably handle it first.

The 8259 PIC provides several priority schemes, but the PC BIOS initializes the 8259 to use *fixed* priority. When using fixed priorities, the device on IRQ 0 (the timer) has the highest priority and the device on IRQ 7 has the lowest priority. Therefore, the 8259 in the PC (running DOS) always resolves conflicts in this manner. If you *were* going to hook that nuclear reactor up to your PC, you'd probably want to use the *nonmaskable* interrupt since it has a higher priority than anything provided by the 8259 (and you can't mask it with a CLI instruction).

17.8 Debugging ISRs

Although writing ISRs can simplify the design of many types of programs, ISRs are almost always very difficult to debug. There are two main reasons ISRs are more difficult than standard applications to debug. First, as mentioned earlier, errant ISRs can modify values the main program uses (or, worse yet, that some *other* program in memory is using) and it is difficult to pin down the source of the error. Second, most debuggers have fits when you attempt to set breakpoints within an ISR.

If your code includes some ISRs and the program seems to be misbehaving and you cannot immediately see the reason, you should immediately suspect interference by the ISR. Many programmers have forgotten about ISRs appearing in their code and have spent weeks attempting to locate a bug in their non-ISR code, only to discover the problem was with the ISR. Always suspect the ISR first. Generally, ISRs are short and you can

quickly eliminate the ISR as the cause of your problem before trying to track the bug down elsewhere.

Debuggers often have problems because they are not reentrant or they call BIOS or DOS (that are not reentrant) so if you set a breakpoint in an ISR that has interrupted BIOS or DOS and the debugger calls BIOS or DOS, the system may crash because of the reentrancy problems. Fortunately, most modern debuggers have a *remote* debugging mode that lets you connect a terminal or another PC to a serial port and execute the debug commands on that second display and keyboard. Since the debugger talks directly to the serial chip, it avoids calling BIOS or DOS and avoids the reentrancy problems. Of course, this doesn't help much if you're writing a *serial* ISR, but it works fine with most other programs.

A big problem when debugging interrupt service routines is that the system crashes immediately after you patch the interrupt vector. If you do not have a remote debugging facility, the best approach to debug this code is to strip the ISR to its bare essentials. This might be the code that simply passes control on to the next ISR in the interrupt chain (if applicable). Then add one section of code at a time back to your ISR until the ISR fails.

Of course, the best debugging strategy is to write code that doesn't have any bugs. While this is not a practical solution, one thing you can do is attempt to do as little as possible in the ISR. Simply read or write the device's data and buffer any inputs for the main program to handle later. The smaller your ISR is, the less complex it is, the higher the probability is that it will not contain any bugs.

Debugging ISRs, unfortunately, is not easy and it is not something you can learn right out of a book. It takes lots of experience and you will need to make a lot of mistakes. There is unfortunately, but there is no substitute for experience when debugging ISRs.

17.9 Summary

This chapter discusses three phenomena occurring in PC systems: interrupts (hardware), traps, and exceptions. An interrupt is an asynchronous procedure call the CPU generates in response to an external hardware signal. A trap is a programmer-supplied call to a routine and is a special form of a procedure call. An exception occurs when a program executes and instruction that generates some sort of error. For additional details, see

- "Interrupts, Traps, and Exceptions" on page 995.

When an interrupt, trap, or exception occurs, the 80x86 CPU pushes the flags and transfers control to an *interrupt service routine* (ISR). The 80x86 supports an *interrupt vector table* that provides segmented addresses for up to 256 different interrupts. When writing your own ISR, you need to store the address of you ISR in an appropriate location in the interrupt vector table to activate that ISR. Well-behaved programs also save the original interrupt vector value so they can restore it when they terminate. For the details, see

- "80x86 Interrupt Structure and Interrupt Service Routines (ISRs)" on page 996

A *trap*, or *software interrupt*, is nothing more than the execution of an 80x86 "int n" instruction. Such an instruction transfers control to the ISR whose vector appears in the nth entry in the interrupt vector table. Generally, you would use a trap to call a routine in a resident program appearing somewhere in memory (like DOS or BIOS). For more information, see

- "Traps" on page 999

An exception occurs whenever the CPU executes an instruction and that instruction is illegal or the execution of that instruction generates some sort of error (like division by zero). The 80x86 provides several built-in exceptions, although this text only deals with the exceptions available in real mode. For the details, see

- "Exceptions" on page 1000

- “Divide Error Exception (INT 0)” on page 1000
- “Single Step (Trace) Exception (INT 1)” on page 1000
- “Breakpoint Exception (INT 3)” on page 1001
- “Overflow Exception (INT 4/INTO)” on page 1001
- “Bounds Exception (INT 5/BOUND)” on page 1001
- “Invalid Opcode Exception (INT 6)” on page 1004
- “Coprocessor Not Available (INT 7)” on page 1004

The PC provides hardware support for up to 15 vectored interrupts using a pair of 8259A programmable interrupt controller chips (PICs). Devices that normally generate hardware interrupts include a timer, the keyboard, serial ports, parallel ports, disk drives, sound cards, the real time clock, and the FPU. The 80x86 lets you enable and disable all *maskable* interrupts with the cli and sti instructions. The PIC also lets you individually mask the devices that can interrupt the system. However, the 80x86 provides a special *nonmaskable* interrupt that has a higher priority than the other hardware interrupts and cannot be disabled by a program. For more details on these hardware interrupts, see

- “Hardware Interrupts” on page 1004
- “The 8259A Programmable Interrupt Controller (PIC)” on page 1005
- “The Timer Interrupt (INT 8)” on page 1007
- “The Keyboard Interrupt (INT 9)” on page 1008
- “The Serial Port Interrupts (INT 0Bh and INT 0Ch)” on page 1008
- “The Parallel Port Interrupts (INT 0Dh and INT 0Fh)” on page 1008
- “The Diskette and Hard Drive Interrupts (INT 0Eh and INT 76h)” on page 1009
- “The Real-Time Clock Interrupt (INT 70h)” on page 1009
- “The FPU Interrupt (INT 75h)” on page 1009
- “Nonmaskable Interrupts (INT 2)” on page 1009
- “Other Interrupts” on page 1009

Interrupt service routines that you write may need to coexist with other ISRs in memory. In particular, you may not be able to simply replace an interrupt vector with the address of your ISR and let your ISR take over from there. Often, you will need to create an *interrupt chain* and call the previous ISR in the interrupt chain once you are done processing the interrupt. To see why you create interrupt chains, and to learn how to create them, see

- “Chaining Interrupt Service Routines” on page 1010

With interrupts comes the possibility of *reentrancy*. that is, the possibility that a routine might be interrupt and called again before the first call finished execution. This chapter introduces the concept of reentrancy and gives some examples that demonstrate problems with nonreentrant code. For details, see

- “Reentrancy Problems” on page 1012

The whole purpose of an interrupt driven system is to improve the efficiency of that system. Therefore, it should come as no surprise that ISRs should be as efficient as possible. This chapter discusses why interrupt driven I/O systems can be more efficient and contrasts interrupt driven I/O with *polled* I/O. However, interrupts can cause problems if the corresponding ISR is too slow. Therefore, programmers who write ISRs need to be aware of such parameters as *interrupt service time, frequency of interrupts,* and *interrupt latency*. To learn about these concepts, see

- “The Efficiency of an Interrupt Driven System” on page 1014
- “Interrupt Driven I/O vs. Polling” on page 1014
- “Interrupt Service Time” on page 1015
- “Interrupt Latency” on page 1016

If multiple interrupts occur simultaneously, the CPU must decide which interrupt to handle first. The 8259 PIC and the PC use a prioritized interrupt scheme assigning the highest priority to the timer and work down from there. The 80x86 always processes the interrupt with the highest priority first. For more details, see

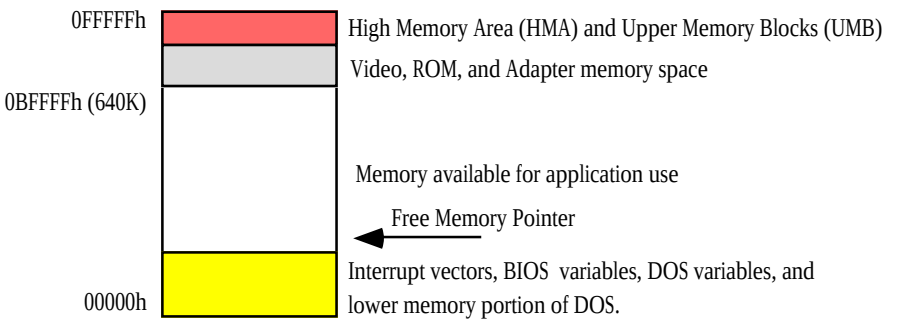
- “Prioritized Interrupts” on page 1020

Most MS-DOS applications are *transient*. They load into memory, execute, terminate, and DOS uses the memory allocated to the application for the next program the user executes. Resident programs follow these same rules, except for the last. A resident program, upon termination, does not return all memory back to DOS. Instead, a portion of the program remains *resident*, ready to be reactivated by some other program at a future time.

Resident programs, also known as *terminate and stay resident programs* or *TSRs*, provide a tiny amount of *multitasking* to an otherwise single tasking operating system. Until Microsoft Windows became popular, resident programs were the most popular way to allow multiple applications to coexist in memory at one time. Although Windows has diminished the need for TSRs for background processing, TSRs are still valuable for writing *device drivers*, *antiviral tools*, and *program patches*. This chapter will discuss the issues you must deal with when writing resident programs.

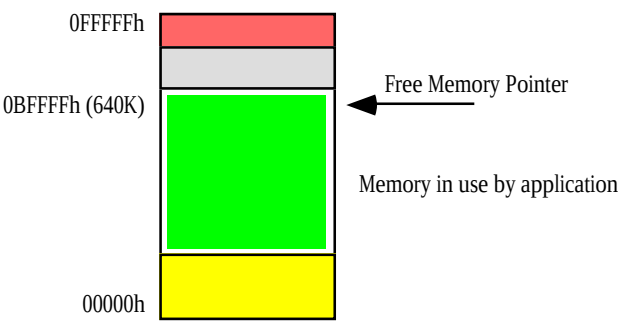
18.1 DOS Memory Usage and TSRs

When you first boot DOS, the memory layout will look something like the following:



DOS Memory Map (no active application)

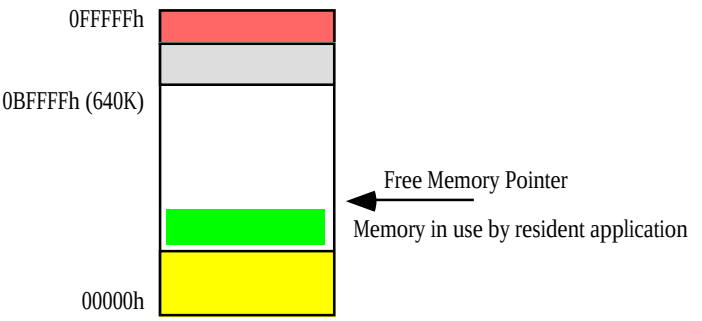
DOS maintains a *free memory pointer* that points the the beginning of the block of free memory. When the user runs an application program, DOS loads this application starting at the address the free memory pointer contains. Since DOS generally runs only a single application at a time, all the memory from the free memory pointer to the end of RAM (0BFFFFh) is available for the application's use:



DOS Memory Map (w/active application)

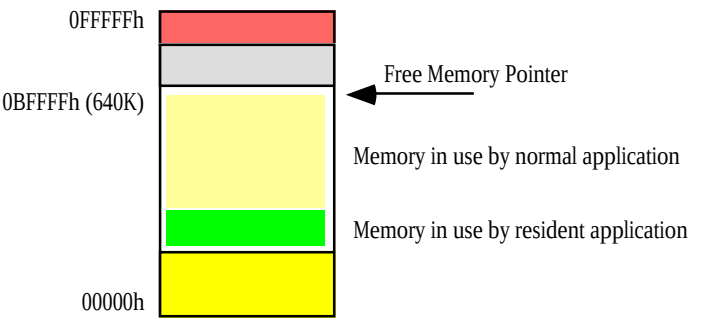
When the program terminates normally via DOS function 4Ch (the Standard Library `exitpgm` macro), MS-DOS reclaims the memory in use by the application and resets the free memory pointer to just above DOS in low memory.

MS-DOS provides a second termination call which is identical to the terminate call with one exception, it does not reset the free memory pointer to reclaim all the memory in use by the application. Instead, this *terminate and stay resident* call frees all but a specified block of memory. The TSR call (ah=31h) requires two parameters, a process termination code in the a1 register (usually zero) and dx must contain the size of the memory block to protect, in paragraphs. When DOS executes this code, it adjusts the free memory pointer so that it points at a location dx*16 bytes above the program's PSP (see "MS-DOS, PC-BIOS, and File I/O" on page 699). This leaves memory looking like this:



DOS Memory Map (w/resident application)

When the user executes a new application, DOS loads it into memory at the new free memory pointer address, protecting the resident program in memory:



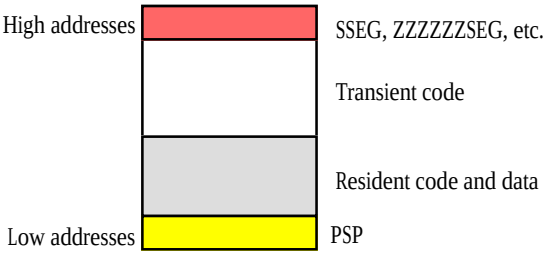
DOS Memory Map (w/resident and normal application)

When this new application terminates, DOS reclaims its memory and readjusts the free memory pointer to its location before running the application – just above the resident program. By using this free memory pointer scheme, DOS can protect the memory in use by the resident program[1].

The trick to using the terminate and stay resident call is to figure out how many paragraphs should remain resident. Most TSRs contain two sections of code: a *resident* portion and a *transient* portion. The transient portion is the data, main program, and support routines that execute when you run the program from the command line. This code will probably never execute again. Therefore, you should not leave it in memory when your program terminates. After all, every byte consumed by the TSR program is one less byte available to other application programs.

The resident portion of the program is the code that remains in memory and provides whatever functions are necessary of the TSR. Since the PSP is usually right before the first byte of program code, to effectively use the DOS TSR call, your program must be organized as follows:

1. Of course, DOS could never protect the resident program from an errant application. If the application decides to write zeros all over memory, the resident program, DOS, and many other memory areas will be destroyed.



Memory Organization for a Resident Program

To use TSRs effectively, you need to organize your code and data so that the resident portions of your program loads into lower memory addresses and the transient portions load into the higher memory addresses. MASM and the Microsoft Linker both provide facilities that let you control the loading order of segments within your code (see "MASM: Directives & Pseudo-Opcodes" on page 355). The simple solution, however, is to put all your resident code and data in a single segment and make sure that this segment appears *first* in every source module of your program. In particular, if you are using the UCR Standard Library SHELL.ASM file, you must make sure that you define your resident segments *before* the include directives for the standard library files. Otherwise MS-DOS will load all the standard library routines *before* your resident segment and that would waste considerable memory. Note that you only need to define your resident segment first, you do not have to place all the resident code and data before the includes. The following will work just fine:

```
ResidentSeg    segment    para public 'resident'
ResidentSeg    ends

EndResident    segment    para public 'EndRes'
EndResident    ends

               .xlist
               include    stdlib.a
               includelib stdlib.lib
               .list


ResidentSeg    segment    para public 'resident'
               assume     cs:ResidentSeg, ds:ResidentSeg

PSP            word       ?             ;This var must be here!

; Put resident code and data here

ResidentSeg    ends


dseg           segment    para public 'data'

; Put transient data here

dseg           ends


cseg           segment    para public 'code'
               assume     cs:cseg, ds:dseg

; Put Transient code here.

cseg           ends
               etc.
```

The purpose of the EndResident segment will become clear in a moment. For more information on DOS memory ordering, see Chapter Six.

Now the only problem is to figure out the size of the resident code, in paragraphs. With your code structured in the manner shown above, determining the size of the resident program is quite easy, just use the following statements to terminate the transient portion of your code (in cseg):

```
                mov     ax, ResidentSeg   ;Need access to ResidentSeg
                mov     es, ax
                mov     ah, 62h           ;DOS Get PSP call.
                int     21h
                mov     es:PSP, bx        ;Save PSP value in PSP variable.

; The following code computes the sixe of the resident portion of the code.
; The EndResident segment is the first segment in memory after resident code.
; The program's PSP value is the segment address of the start of the resident
; block. By computing EndResident-PSP we compute the size of the resident
; portion in paragraphs.

                mov     dx, EndResident   ;Get EndResident segment address.
                sub     dx, bx            ;Subtract PSP.

; Okay, execute the TSR call, preserving only the resident code.

                mov     ax, 3100h         ;AH=31h (TSR), AL=0 (return code).
                int     21h
```

Executing the code above returns control to MS-DOS, preserving your resident code in memory.

There is one final memory management detail to consider before moving on to other topics related to resident programs – accessing data within an resident program. Procedures within a resident program become active in response to a direct call from some other program or a hardware interrupt (see the next section). Upon entry, the resident routine *may* specify that certain registers contain various parameters, but one thing you cannot expect is for the calling code to properly set up the segment registers for you. Indeed, the only segment register that will contain a meaningful value (to the resident code) is the code segment register. Since many resident functions will want to access local data, this means that those functions may need to set up ds or some other segment register(s) upon initial entry. For example, suppose you have a function, count, that simply counts the number of times some other code calls it once it has gone resident. One would thing that the body of this function would contain a single instruction: inc counter. Unfortunately, such an instruction would increment the variable at counter's offset in the current data segment (that is, the segment pointed at by the ds register). It is unlikely that ds would be pointing at the data segment associated with the count procedure. Therefore, you would be incrementing some word in a different segment (probably the caller's data segment). This would produce disastrous results.

There are two solutions to this problem. The first is to put all variables in the code segment (a very common practice in resident sections of code) and use a cs: segment override prefix on all your variables. For example, to increment the counter variable you could use the instruction inc cs:counter. This technique works fine if there are only a few variable references in your procedures. However, it suffers from a few serious drawbacks. First, the segment override prefix makes your instructions larger and slower; this is a serious problem if you access many different variables throughout your resident code. Second, it is easy to forget to place the segment override prefix on a variable, thereby causing the TSR function to wipe out memory in the caller's data segment. Another solution to the segment problem is to change the value in the ds register upon entry to a resident procedure and restore it upon exit. The following code demonstrates how to do this:

```
                push    ds                ;Preserve original DS value.
                push    cs                ;Copy CS's value to DS.
                pop     ds
                inc     Counter           ;Bump the variable's value.
                pop     ds                ;Restore original DS value.
```

Of course, using the cs: segment override prefix is a much more reasonable solution here. However, had the code been extensive and had accessed many local variables, loading ds with cs (assuming you put your variables in the resident segment) would be more efficient.

18.2 Active vs. Passive TSRs

Microsoft identifies two types of TSR routines: active and passive. A passive TSR is one that activates in response to an explicit call from an executing application program. An active TSR is one that responds to a hardware interrupt or one that a hardware interrupt calls.

TSRs are almost always interrupt service routines (see "80x86 Interrupt Structure and Interrupt Service Routines (ISRs)" on page 996). Active TSRs are typically hardware interrupt service routines and passive TSRs are generally trap handlers (see "Traps" on page 999). Although, in theory, it is possible for a TSR to determine the address of a routine in a passive TSR and call that routine directly, the 80x86 trap mechanism is the perfect device for calling such routines, so most TSRs use it.

Passive TSRs generally provide a callable library of routines or extend some DOS or BIOS call. For example, you might want to reroute all characters an application sends to the printer to a file. By patching into the int 17h vector (see "The PC Parallel Ports" on page 1199) you can intercept all characters destined for the printer[2]. Or you could add additional functionality to a BIOS routine by chaining into its interrupt vector. For example, you could add new function calls to the int 10h BIOS video services routine (see "MS-DOS, PC-BIOS, and File I/O" on page 699) by looking for a special value in ah and passing all other int 10h calls on through to the original handler. Another use of a passive TSR is to provide a brand new set of services through a new interrupt vector that the BIOS does not already provide. The mouse services, provided by the mouse.com driver, is a good example of such a TSR.

Active TSRs generally serve one of two functions. They either service a hardware interrupt directly, or they piggyback off the hardware interrupt so they can activate themselves on a periodic basis without an explicit call from an application. *Pop-up* programs are a good example of active TSRs. A pop-up program chains itself into the PC's keyboard interrupt (int 9). Pressing a key activates such a program. The program can read the PC's keyboard port (see "The PC Keyboard" on page 1153) to see if the user is pressing a special key sequence. Should this keysequence appear, the application can save a portion of the screen memory and "pop-up" on the screen, perform some user-requested function, and then restore the screen when done. Borland's Sidekick™ program is an example of an extremely popular TSR program, though many others exist.

Not all active TSRs are pop-ups, though. Certain viruses are good examples of active TSRs. They patch into various interrupt vectors that activate them automatically so they can go about their dastardly deeds. Fortunately, some anti-viral programs are also good examples of active TSRs, they patch into those same interrupt vectors and detect the activities of a virus and attempt to limit the damage the virus may cause.

Note that a TSR may contain both active and passive components. That is, there may be certain routines that a hardware interrupt invokes and others that an application calls explicitly. However, if any routine in a resident program is active, we'll claim that the entire TSR is active.

The following program is a short example of a TSR that provides both active and passive routines. This program patches into the int 9 (keyboard interrupt) and int 16h (keyboard trap) interrupt vectors. Every time the system generates a keyboard interrupt, the active routine (int 9) increments a counter. Since the keyboard usually generates two keyboard interrupts per keystroke, dividing this value by two produces the approximate number of keys typed since starting the TSR[3]. A passive routine, tied into the int 16h vector, returns the number of keystrokes to the calling program. The following code provides two programs, the TSR and a short application to display the number of keystrokes since the TSR started running.

```
; This is an example of an active TSR that counts keyboard interrupts
; once activated.

; The resident segment definitions must come before everything else.
```

2. Assuming the application uses DOS or BIOS to print the characters and does not talk directly to the printer port itself.
3. It is not an exact count because some keys generate more than two keyboard interrupts.

```
ResidentSeg    segment    para public 'Resident'
ResidentSeg    ends

EndResident    segment    para public 'EndRes'
EndResident    ends

               .xlist
               include    stdlib.a
               includelib stdlib.lib
               .list


; Resident segment that holds the TSR code:

ResidentSeg    segment    para public 'Resident'
               assume     cs:ResidentSeg, ds:nothing

; The following variable counts the number of keyboard interrupts

KeyIntCnt      word       0

; These two variables contain the original INT 9 and INT 16h
; interrupt vector values:

OldInt9        dword      ?
OldInt16       dword      ?


; MyInt9-      The system calls this routine every time a keyboard
;              interrupt occus. This routine increments the
;              KeyIntCnt variable and then passes control on to the
;              original Int9 handler.

MyInt9         proc       far
               inc        ResidentSeg:KeyIntCnt
               jmp        ResidentSeg:OldInt9
MyInt9         endp




; MyInt16-     This is the passive component of this TSR. An
;              application explicitly calls this routine with an
;              INT 16h instruction. If AH contains 0FFh, this
;              routine returns the number of keyboard interrupts
;              in the AX register. If AH contains any other value,
;              this routine passes control to the original INT 16h
;              (keyboard trap) handler.

MyInt16        proc       far
               cmp        ah, 0FFh
               je         ReturnCnt
               jmp        ResidentSeg:OldInt16;Call original handler.

; If AH=0FFh, return the keyboard interrupt count

ReturnCnt:     mov        ax, ResidentSeg:KeyIntCnt
               iret
MyInt16        endp


ResidentSeg    ends



cseg           segment    para public 'code'
               assume     cs:cseg, ds:ResidentSeg

Main           proc
               meminit

               mov        ax, ResidentSeg
               mov        ds, ax
```

```
                mov     ax, 0
                mov     es, ax

                print
                byte    "Keyboard interrupt counter TSR program",cr,lf
                byte    "Installing....",cr,lf,0

; Patch into the INT 9 and INT 16 interrupt vectors. Note that the
; statements above have made ResidentSeg the current data segment,
; so we can store the old INT 9 and INT 16 values directly into
; the OldInt9 and OldInt16 variables.

                cli                     ;Turn off interrupts!
                mov     ax, es:[9*4]
                mov     word ptr OldInt9, ax
                mov     ax, es:[9*4 + 2]
                mov     word ptr OldInt9+2, ax
                mov     es:[9*4], offset MyInt9
                mov     es:[9*4+2], seg ResidentSeg

                mov     ax, es:[16h*4]
                mov     word ptr OldInt16, ax
                mov     ax, es:[16h*4 + 2]
                mov     word ptr OldInt16+2, ax
                mov     es:[16h*4], offset MyInt16
                mov     es:[16h*4+2], seg ResidentSeg
                sti                     ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte    "Installed.",cr,lf,0

                mov     ah, 62h         ;Get this program's PSP
                int     21h             ; value.

                mov     dx, EndResident ;Compute size of program.
                sub     dx, bx
                mov     ax, 3100h       ;DOS TSR command.
                int     21h
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

Here's the application that calls MyInt16 to print the number of keystrokes:

```
; This is the companion program to the keycnt TSR.
; This program calls the "MyInt16" routine in the TSR to
; determine the number of keyboard interrupts. It displays
; the approximate number of keystrokes (keyboard ints/2)
; and quits.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

cseg            segment    para public 'code'
                assume     cs:cseg, ds:nothing

Main            proc
                meminit

                print
```

```
                byte        "Approximate number of keys pressed: ",0
                mov         ah, 0FFh
                int         16h
                shr         ax, 1             ;Must divide by two.
                putu
                putcr
                ExitPgm

Main            endp
cseg            ends

sseg            segment     para stack 'stack'
stk             db          1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

18.3 Reentrancy

One big problem with active TSRs is that their invocation is asynchronous. They can activate at the touch of a keystroke, timer interrupt, or via an incoming character on the serial port, just to name a few. Since they activate on a hardware interrupt, the PC could have been executing just about any code when the interrupt came along. This isn't a problem unless the TSR itself decides to call some foreign code, such as DOS, a BIOS routine, or some other TSR. For example, the main application may be making a DOS call when a timer interrupt activates a TSR, interrupting the call to DOS while the CPU is still executing code inside DOS. If the TSR attempts to make a call to DOS at this point, then this will *reenter* DOS. Of course, DOS is not reentrant, so this creates all kinds of problems (usually, it hangs the system). When writing active TSRs that call other routines besides those provided directly in the TSR, you must be aware of possible reentrancy problems.

Note that passive TSRs never suffer from this problem. Indeed, any TSR routine you call passively will execute in the caller's environment. Unless some other hardware ISR or active TSR makes the call to your routine, you do not need to worry about reentrancy with passive routines. However, reentrancy is an issue for active TSR routines and passive routines that active TSRs call.

18.3.1 Reentrancy Problems with DOS

DOS is probably the biggest sore point to TSR developers. DOS is not reentrant yet DOS contains many services a TSR might use. Realizing this, Microsoft has added some support to DOS to allow TSRs to see if DOS is currently active. After all, reentrancy is only a problem if you call DOS while it is already active. If it isn't already active, you can certainly call it from a TSR with no ill effects.

MS-DOS provides a special one-byte flag (InDOS) that contains a zero if DOS is currently active and a non-zero value if DOS is already processing an application request. By testing the InDOS flag your TSR can determine if it can safely make a DOS call. If this flag is zero, you can always make the DOS call. If this flag contains one, you may not be able to make the DOS call. MS-DOS provides a function call, *Get InDOS Flag Address*, that returns the address of the InDOS flag. To use this function, load `ah` with 34h and call DOS. DOS will return the address of the InDOS flag in `es:bx`. If you save this address, your resident programs will be able to test the InDOS flag to see if DOS is active.

Actually, there are two flags you should test, the InDOS flag and the *critical error flag* (criterr). Both of these flags should contain zero before you call DOS from a TSR. In DOS version 3.1 and later, the critical error flag appears in the byte just before the InDOS flag.

So what should you do if these flags aren't both zero? It's easy enough to say "hey, come back and do this stuff later when MS-DOS returns back to the user program." But how do you do this? For example, if a keyboard interrupt activates your TSR and you pass control on to the real keyboard handler because DOS is busy, you can't expect your TSR to be magically restarted later on when DOS is no longer active.

The trick is to patch your TSR into the timer interrupt as well as the keyboard interrupt. When the keystroke interrupt wakes your TSR and you discover that DOS is busy, the keyboard ISR can simply set a flag to tell itself to try again later; then it passes control to the original keyboard handler. In the meantime, a timer ISR you've written is constantly checking this flag you've created. If the flag is clear, it simply passes control on to the original timer interrupt handler, if the flag is set, then the code checks the InDOS and CritErr flags. If these guys say that DOS is busy, the timer ISR passes control on to the original timer handler. Shortly after DOS finishes whatever it was doing, a timer interrupt will come along and detect that DOS is no longer active. Now your ISR can take over and make any necessary calls to DOS that it wants. Of course, once your timer code determines that DOS is not busy, it should clear the "I want service" flag so that future timer interrupts don't inadvertently restart the TSR.

There is only one problem with this approach. There are certain DOS calls that can take an indefinite amount of time to execute. For example, if you call DOS to read a key from the keyboard (or call the Standard Library's `getc` routine that calls DOS to read a key), it could be *hours*, *days*, or even longer before somebody actually bothers to press a key. Inside DOS there is a loop that waits until the user actually presses a key. And until the user presses some key, the InDOS flag is going to remain non-zero. If you've written a timer-based TSR that is buffering data every few seconds and needs to write the results to disk every now and then, you will overflow your buffer with new data if you wait for the user, who just went to lunch, to press a key in DOS' command.com program.

Luckily, MS-DOS provides a solution to this problem as well – the idle interrupt. While MS-DOS is in an indefinite loop wait for an I/O device, it continually executes an `int 28h` instruction. By patching into the int 28h vector, your TSR can determine when DOS is sitting in such a loop. When DOS executes the int 28h instruction, it is safe to make any DOS call whose function number (the value in `ah`) is greater than 0Ch.

So if DOS is busy when your TSR wants to make a DOS call, you must use either a timer interrupt or the idle interrupt (int 28h) to activate the portion of your TSR that must make DOS calls. One final thing to keep in mind is that *whenever you test or modify any of the above mentioned flags, you are in a critical section*. Make sure the interrupts are off. If not, your TSR make activate two copies of itself or you may wind up entering DOS at the same time some other TSR enters DOS.

An example of a TSR using these techniques will appear a little later, but there are some additional reentrancy problems we need to discuss first.

18.3.2 Reentrancy Problems with BIOS

DOS isn't the only non-reentrant code a TSR might want to call. The PC's BIOS routines also fall into this category. Unfortunately, BIOS doesn't provide an "InBIOS" flag or a multiplex interrupt. You will have to supply such functionality yourself.

The key to preventing reentering a BIOS routine you want to call is to use a *wrapper*. A wrapper is a short ISR that patches into an existing BIOS interrupt specifically to manipulate an InUse flag. For example, suppose you need to make an int 10h (video services) call from within your TSR. You could use the following code to provide an "Int10InUse" flag that your TSR could test:

```
MyInt10         proc        far
                inc         cs:Int10InUse
                pushf
                call        cs:OldInt10
                dec         cs:Int10InUse
                iret
MyInt10         endp
```

Assuming you've initialized the Int10InUse variable to zero, the in use flag will contain zero when it is safe to execute an int 10h instruction in your TSR, it will contain a non-zero value when the interrupt 10h handler is busy. You can use this flag like the InDOS flag to defer the execution of your TSR code.

Like DOS, there are certain BIOS routines that may take an indefinite amount of time to complete. Reading a key from the keyboard buffer, reading or writing characters on the serial port, or printing characters to the printer are some examples. While, in some cases, it is possible to create a wrapper that lets your TSR activate itself while a BIOS routine is executing one of these polling loops, there is probably no benefit to doing so. For example, if an application program is waiting for the printer to take a character before it sends another to printer, having your TSR preempt this and attempt to send a character to the printer won't accomplish much (other than scramble the data sent to the print). Therefore, BIOS wrappers generally don't worry about *indefinite postponement* in a BIOS routine.

5, 8, 9, D, E, 10, 13, 16, 17, 21, 28

If you run into problems with your TSR code and certain application programs, you may want to place wrappers around the following interrupts to see if this solves your problem: int 5, int 8, int 9, int B, int C, int D, int E, int 10, int 13, int 14, int 16, or int 17. These are common culprits when TSR problems develop.

18.3.3 Reentrancy Problems with Other Code

Reentrancy problems occur in other code you might call as well. For example, consider the UCR Standard Library. The UCR Standard Library is not reentrant. This usually isn't much of a problem for a couple of reasons. First, most TSRs do *not* call Standard Library subroutines. Instead, they provide results that normal applications can use; those applications use the Standard Library routines to manipulate such results. A second reason is that were you to include some Standard Library routines in a TSR, the application would have a *separate* copy of the library routines. The TSR might execute an strcmp instruction while the application is in the middle of an strcmp routine, *but these are not the same routines!* The TSR is not reentering the application's code, it is executing a separate routine.

However, many of the Standard Library functions make DOS or BIOS calls. Such calls do not check to see if DOS or BIOS is already active. Therefore, calling many Standard Library routines from within a TSR may cause you to reenter DOS or BIOS.

One situation does exist where a TSR could reenter a Standard Library routine. Suppose your TSR has both passive and active components. If the main application makes a call to a passive routine in your TSR and that routine call a Standard Library routine, there is the possibility that a system interrupt could interrupt the Standard Library routine and the active portion of the TSR reenter that same code. Although such a situation would be extremely rare, you should be aware of this possibility.

Of course, the best solution is to avoid using the Standard Library within your TSRs. If for no other reason, the Standard Library routines are quite large and TSRs should be as small as possible.

18.4 The Multiplex Interrupt (INT 2Fh)

When installing a passive TSR, or an active TSR with passive components, you will need to choose some interrupt vector to patch so other programs can communicate with your passive routines. You could pick an interrupt vector almost at random, say int 84h, but this could lead to some compatibility problems. What happens if someone else is already using that interrupt vector? Sometimes, the choice of interrupt vector is clear. For example, if your passive TSR is extended the int 16h keyboard services, it makes sense to patch in to the int 16h vector and add additional functions above and beyond those already provided by the BIOS. On the other hand, if you are creating a driver for some brand new device for the PC, you probably would not want to piggyback the support functions for this device on some other interrupt. Yet arbitrarily picking an unused interrupt vector is risky; how many other programs out there decided to do the

same thing? Fortunately, MS-DOS provides a solution: the multiplex interrupt. Int 2Fh provides a general mechanism for installing, testing the presence of, and communicating with a TSR.

To use the multiplex interrupt, an application places an identification value in ah and a function number in al and then executes an int 2Fh instruction. Each TSR in the int 2Fh chain compares the value in ah against its own unique identifier value. If the values match, the TSR process the command specified by the value in the al register. If the identification values do not match, the TSR passes control to the next int 2Fh handler in the chain.

Of course, this only reduces the problem somewhat, it doesn't eliminate it. Sure, we don't have to guess an interrupt vector number at random, but we still have to choose a random identification number. After all, it seems reasonable that we must choose this number before designing the TSR and any applications that call it, after all, how will the applications know what value to load into ah if we dynamically assign this value when the TSR goes resident?

Well, there is a little trick we can play to dynamically assign TSR identifiers *and* let any interested applications determine the TSR's ID. By convention, function zero is the "Are you there?" call. An application should always execute this function to determine if the TSR is actually present in memory before making any service requests. Normally, function zero returns a zero in al if the TSR is *not* present, it returns 0FFh if it is present. However, when this function returns 0FFh it only tells you that *some* TSR has responded to your query; it does not guarantee that the TSR you are interested in is actually present in memory. However, by extending the convention somewhat, it is very easy to verify the presence of the desired TSR. Suppose the function zero call also returns a pointer to a unique identification string in the es:di registers. Then the code testing for the presence of a specific TSR could test this string when the int 2Fh call detects the presence of a TSR. the following code segment demonstrates how a TSR could determine if a TSR identified as "Randy's INT 10h Extension" is present in memory; this code will also determine the unique identification code for that TSR, for future reference:

```
; Scan through all the possible TSR IDs. If one is installed, see if
; it's the TSR we're interested in.

                mov         cx, 0FFh            ;This will be the ID number.
IDLoop:         mov         ah, cl              ;ID -> AH.
                push        cx                  ;Preserve CX across call
                mov         al, 0               ;Test presence function code.
                int         2Fh                 ;Call multiplex interrupt.
                pop         cx                  ;Restore CX.
                cmp         al, 0               ;Installed TSR?
                je          TryNext             ;Returns zero if none there.
                strcmpl                         ;See if it's the one we want.
                byte        "Randy's INT "
                byte        "10h Extension",0
                je          Success             ;Branch off if it is ours.
TryNext:        loop        IDLoop              ;Otherwise, try the next one.
                jmp         NotInstalled        ;Failure if we get to this point.

Success:        mov         FuncID, cl          ;Save function result.
                 .
                 .
                 .
```

If this code succeeds, the variable FuncId contains the identification value for resident TSR. If it fails, the application program probably needs to abort, or otherwise ensure that it never calls the missing TSR.

The code above lets an application easily detect the presence of and determine the ID number for a specific TSR. The next question is "How do we pick the ID number for the TSR in the first place?" The next section will address that issue, as well as how the TSR must respond to the multiplex interrupt.

18.5 Installing a TSR

Although we've already discussed how to make a program go resident (see "DOS Memory Usage and TSRs" on page 1025), there are a few aspects to installing a TSR that we need to address. First, what hap-

pens if a user installs a TSR and then tries to install it a second time without first removing the one that is already resident? Second, how can we assign a TSR identification number that won't conflict with a TSR that is already installed? This section will address these issues.

The first problem to address is an attempt to reinstall a TSR program. Although one could imagine a type of TSR that allows multiple copies of itself in memory at one time, such TSRs are few and far in-between. In most cases, having multiple copies of a TSR in memory will, at best, waste memory and, at worst, crash the system. Therefore, unless you are specifically written a TSR that allows multiple copies of itself in memory at one time, you should check to see if the TSR is installed before actually installing it. This code is identical to the code an application would use to see if the TSR is installed, the only difference is that the TSR should print a nasty message and refuse to go TSR if it finds a copy of itself already installed in memory. The following code does this:

```
                mov         cx, 0FFh
SearchLoop:     mov         ah, cl
                push        cx
                mov         al, 0
                int         2Fh
                pop         cx
                cmp         al, 0
                je          TryNext
                strcmpl
                byte        "Randy's INT "
                byte        "10h Extension",0
                je          AlreadyThere
TryNext:        loop        SearchLoop
                jmp         NotInstalled

AlreadyThere:   print
                byte        "A copy of this TSR already exists in memory",cr,lf
                byte        "Aborting installation process.",cr,lf,0
                ExitPgm
                 .
                 .
                 .
```

In the previous section, you saw how to write some code that would allow an application to determine the TSR ID of a specific resident program. Now we need to look at how to dynamically choose an identification number for the TSR, one that does not conflict with any other TSRs. This is yet another modification to the scanning loop. In fact, we can modify the code above to do this for us. All we need to do is save away some ID value that does not does not have an installed TSR. We need only add a few lines to the above code to accomplish this:

```
                mov         FuncID, 0           ;Initialize FuncID to zero.
                mov         cx, 0FFh
SearchLoop:     mov         ah, cl
                push        cx
                mov         al, 0
                int         2Fh
                pop         cx
                cmp         al, 0
                je          TryNext
                strcmpl
                byte        "Randy's INT "
                byte        "10h Extension",0
                je          AlreadyThere
                loop        SearchLoop
                jmp         NotInstalled

; Note:  presumably DS points at the resident data segment that contains
;        the FuncID variable. Otherwise you must modify the following to
;        point some segment register at the segment containing FuncID and
;        use the appropriate segment override on FuncID.

TryNext:        mov         FuncID, cl          ;Save possible function ID if this
                loop        SearchLoop          ; identifier is not in use.
                jmp         NotInstalled

AlreadyThere:   print
```

```
                byte        "A copy of this TSR already exists in memory",cr,lf
                byte        "Aborting installation process.",cr,lf,0
                ExitPgm

NotInstalled:   cmp         FuncID, 0         ;If there are no available IDs, this
                jne         GoodID            ; will still contain zero.
                print
                byte        "There are too many TSRs already installed.",cr,lf
                byte        "Sorry, aborting installation process.",cr,lf,0
                ExitPgm

GoodID:
```

If this code gets to label "GoodID" then a previous copy of the TSR is not present in memory and the FuncID variable contains an unused function identifier.

Of course, when you install your TSR in this manner, you must not forget to patch your interrupt 2Fh handler into the int 2Fh chain. Also, you have to write an interrupt 2Fh handler to process int 2Fh calls. The following is a very simple multiplex interrupt handler for the code we've been developing:

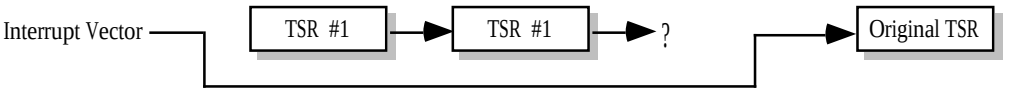
```
FuncID          byte        0                 ;Should be in resident segment.
OldInt2F        dword       ?                 ; Ditto.

MyInt2F         proc        far
                cmp         ah, cs:FuncID     ;Is this call for us?
                je          ItsUs
                jmp         cs:OldInt2F       ;Chain to previous guy, if not.

; Now decode the function value in AL:

ItsUs:          cmp         al, 0             ;Verify presence call?
                jne         TryOtherFunc
                mov         al, 0FFh          ;Return "present" value in AL.
                lesi        IDString          ;Return pointer to string in es:di.
                iret                          ;Return to caller.
IDString        byte        ""Randy's INT "
                byte        "10h Extension",0

; Down here, handle other multiplex requests.
; This code doesn't offer any, but here's where they would go.
; Just test the value in AL to determine which function to execute.

TryOtherFunc:
                 .
                 .
                 .
                iret
MyInt2F         endp
```

18.6 Removing a TSR

Removing a TSR is quite a bit more difficult that installing one. There are three things the removal code must do in order to properly remove a TSR from memory: first, it needs to stop any pending activities (e.g., the TSR may have some flags set to start some activity at a future time); second it needs to restore all interrupt vectors to their former values; third, it needs to return all reserved memory back to DOS so other applications can make use of it. The primary difficulty with these three activities is that it is not always possible to properly restore the interrupt vectors.

If your TSR removal code simply restores the old interrupt vector values, you may create a really big problem. What happens if the user runs some other TSRs after running yours and they patch into the same interrupt vectors as your TSR? This would produce interrupt chains that look something like the following:

If you restore the interrupt vector with your original value, you will create the following:



This effectively disables the TSRs that chain into your code. Worse yet, this only disables the interrupts that those TSRs have in common with your TSR. the other interrupts those TSRs patch into are still active. Who knows how those interrupts will behave under such circumstances?

One solution is to simply print an error message informing the user that they cannot remove this TSR until they remove all TSRs installed prior to this one. This is a common problem with TSRs and most DOS users who install and remove TSRs should be comfortable with the fact that they must remove TSRs in the reverse order that they install them.

It would be tempting to suggest a new convention that TSRs should obey; perhaps if the function number is 0FFh, a TSR should store the value in `es:bx` away in the interrupt vector specified in `cl`. This would allow a TSR that would like to remove itself to pass the address of its original interrupt handler to the previous TSR in the chain. There are only three problems with this approach: first, almost no TSRs in existence currently support this feature, so it would be of little value; second, some TSRs might use function 0FFh for something else, calling them with this value, *even if you knew their ID number*, could create a problem; finally, just because you've removed the TSR from the interrupt chain doesn't mean you can (truly) free up the memory the TSR uses. DOS' memory management scheme (the free pointer business) works like a stack. If there are other TSRs installed above yours in memory, most applications wouldn't be able to use the memory freed up by removing your TSR anyway.

Therefore, we'll also adopt the strategy of simply informing the user that they cannot remove a TSR if there are others installed in shared interrupt chains. Of course, that does bring up a good question, how can we determine if there are other TSRs chained in to our interrupts? Well, this isn't so hard. We know that the 80x86's interrupt vectors should still be pointing at our routines if we're the last TSR run. So all we've got to do is compare the patched interrupt vectors against the addresses of our interrupt service routines. If they *all* match, then we can safely remove our TSR from memory. If only one of them does not match, then we cannot remove the TSR from memory. The following code sequence tests to see if it is okay to detach a TSR containing ISRs for int 2fH and int 9:

```
; OkayToRmv-  This routine returns the carry flag set if it is okay to
;             remove the current TSR from memory. It checks the interrupt
;             vectors for int 2F and int 9 to make sure they
;             are still pointing at our local routines.
;             This code assumes DS is pointing at the resident code's
;             data segment.

OkayToRmv     proc      near
              push      es
              mov       ax, 0             ;Point ES at interrupt vector
              mov       es, ax            ; table.
              mov       ax, word ptr OldInt2F
              cmp       ax, es:[2fh*4]
              jne       CantRemove
              mov       ax, word ptr OldInt2F+2
              cmp       ax, es:[2Fh*4 + 2]
              jne       CantRemove

              mov       ax, word ptr OldInt9
              cmp       ax, es:[9*4]
              jne       CantRemove
              mov       ax, word ptr OldInt9+2
              cmp       ax, es:[9*4 + 2]
              jne       CantRemove

; We can safely remove this TSR from memory.

              stc
              pop       es
              ret
```

```
' Someone else is in the way, we cannot remove this TSR.

CantRemove:    clc
               pop        es
               ret
OkayToRmv      endp
```

Before the TSR attempts to remove itself, it should call a routine like this one to see if removal is possible.

Of course, the fact that no other TSR has chained into the same interrupts does *not* guarantee that there are not TSRs above yours in memory. However, removing the TSR in that case will not crash the system. True, you may not be able to reclaim the memory the TSR is using (at least until you remove the other TSRs), but at least the removal will not create complications.

To remove the TSR from memory requires two DOS calls, one to free the memory in use by the TSR and one to free the memory in use by the environment area assigned to the TSR. To do this, you need to make the DOS deallocation call (see "MS-DOS, PC-BIOS, and File I/O" on page 699). This call requires that you pass the segment address of the block to release in the `es` register. For the TSR program itself, you need to pass the address of the TSR's PSP. This is one of the reasons a TSR needs to save its PSP when it first installs itself. The other free call you must make frees the space associated with the TSR's *environment block*. The address of this block is at offset 2Ch in the PSP. So we should probably free it first. The following calls handle the job of free the memory associated with a TSR:

```
; Presumably, the PSP variable was initialized with the address of this
; program's PSP before the terminate and stay resident call.

               mov        es, PSP
               mov        es, es:[2Ch]      ;Get address of environment block.
               mov        ah, 49h           ;DOS deallocate block call.
               int        21h

               mov        es, PSP           ;Now free the program's memory
               mov        ah, 49h           ; space.
               int        21h
```

Some poorly-written TSRs provide no facilities to allow you to remove them from memory. If someone wants remove such a TSR, they will have to reboot the PC. Obviously, this is a poor design. Any TSR you design for anything other than a quick test should be capable of removing itself from memory. The multiplex interrupt with function number one is often used for this purpose. To remove a TSR from memory, some application program passes the TSR ID and a function number of one to the TSR. If the TSR can remove itself from memory, it does so and returns a value denoting success. If the TSR cannot remove itself from memory, it returns some sort of error condition.

Generally, the removal program is the TSR itself with a special parameter that tells it to remove the TSR currently loaded into memory. A little later this chapter presents an example of a TSR that works precisely in this fashion (see "A Keyboard Monitor TSR" on page 1041).

18.7 Other DOS Related Issues

In addition to reentrancy problems with DOS, there are a few other issues your TSRs must deal with if they are going to make DOS calls. Although your calls might not cause DOS to reenter itself, it is quite possible for your TSR's DOS calls to disturb data structures in use by an executing application. These data structures include the application's stack, PSP, disk transfer area (DTA), and the DOS extended error information record.

When an active or passive TSR gains control of the CPU, it is operating in the environment of the main (foreground) application. For example, the TSR's return address and any values it saves on the stack are pushed onto the application's stack. If the TSR does not use much stack space, this is fine, it need not switch stacks. However, if the TSR consumes considerable amounts of stack space because of recursive

calls or the allocation of local variables, the TSR should save the application's ss and sp values and switch to a local stack. Before returning, of course, the TSR should switch back to the foreground application's stack.

Likewise, if the TSR execute's DOS' *get psp address* call, DOS returns the address of the foreground application's PSP, not the TSR's PSP[4]. The PSP contains several important address that DOS uses in the event of an error. For example, the PSP contains the address of the termination handler, ctrl-break handler, and critical error handler. If you do not switch the PSP from the foreground application to the TSR's and one of the exceptions occurs (e.g., someone hits control-break or a disk error occurs), the handler associated with the application may take over. Therefore, when making DOS calls that can result in one of these conditions, you need to switch PSPs. Likewise, when your TSR returns control to the foreground application, it must restore the PSP value. MS-DOS provides two functions that get and set the current PSP address. The DOS *Set PSP* call (ah=51h) sets the current program's PSP address to the value in the bx register. The DOS *Get PSP* call (ah=50h) returns the current program's PSP address in the bx register. Assuming the transient portion of your TSR has saved it's PSP address in the variable PSP, you switch between the TSR's PSP and the foreground application's PSP as follows:

```
; Assume we've just entered the TSR code, determined that it's okay to
; call DOS, and we've switch DS so that it points at our local variables.

                mov     ah, 51h         ;Get application's PSP address
                int     21h
                mov     AppPSP, bx      ;Save application's PSP locally.
                mov     bx, PSP         ;Change system PSP to TSR's PSP.
                mov     ah, 50h         ;Set PSP call
                int     21h
                 .
                 .                      ;TSR code
                 .
                mov     bx, AppPSP      ;Restore system PSP address to
                mov     ah, 50h         ; point at application's PSP.
                int     21h

        « clean up and return from TSR »
```

Another global data structure that DOS uses is the *disk transfer area*. This buffer area was used extensively for disk I/O in DOS version 1.0. Since then, the main use for the DTA has been the find first file and find next file functions (see "MS-DOS, PC-BIOS, and File I/O" on page 699). Obviously, if the application is in the middle of using data in the DTA and your TSR makes a DOS call that changes the data in the DTA, you will affect the operation of the foreground process. MS-DOS provides two calls that let you get and set the address of the DTA. The *Get DTA Address* call, with ah=2Fh, returns the address of the DTA in the es:bx registers. The *Set DTA* call (ah=1Ah) sets the DTA to the value found in the ds:dx register pair. With these two calls you can save and restore the DTA as we did for the PSP address above. The DTA is usually at offset 80h in the PSP, the following code preserve's the foreground application's DTA and sets the current DTA to the TSR's at offset PSP:80.

```
; This code makes the same assumptions as the previous example.

                mov     ah, 2Fh         ;Get application DTA
                int     21h
                mov     word ptr AppDTA, bx
                mov     word ptr AppDTA+2, es

                push    ds
                mov     ds, PSP         ;DTA is in PSP
                mov     dx, 80h         ; at offset 80h
                mov     ah, 1ah         ;Set DTA call.
                int     21h
                pop     ds
                 .
                 .                      ;TSR code.
                 .
```

4. This is another reason the transient portion of the TSR must save the PSP address in a resident variable for the TSR.

```
                push    ds
                mov     dx, word ptr AppDTA
                mov     ds, word ptr AppDTA+2
                mov     ax, 1ah         ;Set DTA call.
                int     21h
```

The last issue a TSR must deal with is the extended error information in DOS. If a TSR interrupts a program immediately after DOS returns to that program, there may be some error information the foreground application needs to check in the DOS extended error information. If the TSR makes any DOS calls, DOS may replace this information with the status of the TSR DOS call. When control returns to the foreground application, it may read the extended error status and get the information generated by the TSR DOS call, not the application's DOS call. DOS provides two asymmetrical calls, *Get Extended Error* and *Set Extended Error* that read and write these values, respectively. The call to Get Extended Error returns the error status in the ax, bx, cx, dx, si, di, es, and ds registers. You need to save the registers in a data structure that takes the following form:

```
ExtError        struct
eeAX            word        ?
eeBX            word        ?
eeCX            word        ?
eeDX            word        ?
eeSI            word        ?
eeDI            word        ?
eeDS            word        ?
eeES            word        ?
                word        3 dup (0)       ;Reserved.
ExtError        ends
```

The Set Extended Error call requires that you pass an address to this structure in the ds:si register pair (which is why these two calls are asymmetrical). To preserve the extended error information, you would use code similar to the following:

```
; Save assumptions as the above routines here. Also, assume the error
; data structure is named ERR and is in the same segment as this code.

                push    ds              ;Save ptr to our DS.
                mov     ah, 59h         ;Get extended error call
                mov     bx, 0           ;Required by this call
                int     21h

                mov     cs:ERR.eeDS, ds
                pop     ds              ;Retrieve ptr to our data.
                mov     ERR.eeAX, ax
                mov     ERR.eeBX, bx
                mov     ERR.eeCX, cx
                mov     ERR.eeDX, dx
                mov     ERR.eeSI, si
                mov     ERR.eeDI, di
                mov     ERR.eeES, es
                 .
                 .                      ;TSR code goes here.
                 .
                mov     si, offset ERR  ;DS already points at correct seg.
                mov     ax, 5D0Ah       ;5D0Ah is Set Extended Error code.
                int     21h

        « clean up and quit »
```

18.8 A Keyboard Monitor TSR

The following program extends the keystroke counter program presented a little earlier in this chapter. This particular program monitors keystrokes and each minute writes out data to a file listing the date, time, and approximate number of keystrokes in the last minute.

This program can help you discover how much time you spend typing versus thinking at a display screen[5].

```
; This is an example of an active TSR that counts keyboard interrupts
; once activated. Every minute it writes the number of keyboard
; interrupts that occurred in the previous minute to an output file.
; This continues until the user removes the program from memory.
;
;
; Usage:
;       KEYEVAL filename    -          Begins logging keystroke data to
;                                      this file.
;
;       KEYEVAL REMOVE      -          Removes the resident program from
;                                      memory.
;
;
; This TSR checks to make sure there isn't a copy already active in
; memory. When doing disk I/O from the interrupts, it checks to make
; sure DOS isn't busy and it preserves application globals (PSP, DTA,
; and extended error info). When removing itself from memory, it
; makes sure there are no other interrupts chained into any of its
; interrupts before doing the remove.
;
; The resident segment definitions must come before everything else.

ResidentSeg    segment    para public 'Resident'
ResidentSeg    ends

EndResident    segment    para public 'EndRes'
EndResident    ends

               .xlist
               .286
               include    stdlib.a
               includelib stdlib.lib
               .list


; Resident segment that holds the TSR code:

ResidentSeg    segment    para public 'Resident'
               assume     cs:ResidentSeg, ds:nothing

; Int 2Fh ID number for this TSR:

MyTSRID        byte       0

; The following variable counts the number of keyboard interrupts

KeyIntCnt      word       0

; Counter counts off the number of milliseconds that pass, SecCounter
; counts off the number of seconds (up to 60).

Counter        word       0
SecCounter     word       0

; FileHandle is the handle for the log file:

FileHandle     word       0

; NeedIO determines if we have a pending I/O opearation.

NeedIO         word       0

; PSP is the psp address for this program.

PSP            word       0
```

5. This program is intended for your personal enjoyment only, it is not intended to be used for unethical purposes such as monitoring employees for evaluation purposes.

```
; Variables to tell us if DOS, INT 13h, or INT 16h are busy:

InInt13         byte        0
InInt16         byte        0
InDOSFlag       dword       ?

; These variables contain the original values in the interrupt vectors
; we've patched.

OldInt9         dword       ?
OldInt13        dword       ?
OldInt16        dword       ?
OldInt1C        dword       ?
OldInt28        dword       ?
OldInt2F        dword       ?


; DOS data structures:

ExtErr          struct
eeAX            word        ?
eeBX            word        ?
eeCX            word        ?
eeDX            word        ?
eeSI            word        ?
eeDI            word        ?
eeDS            word        ?
eeES            word        ?
                word        3 dup (0)
ExtErr          ends



XErr            ExtErr      {}              ;Extended Error Status.
AppPSP          word        ?               ;Application PSP value.
AppDTA          dword       ?               ;Application DTA address.


; The following data is the output record. After storing this data
; to these variables, the TSR writes this data to disk.

month           byte        0
day             byte        0
year            word        0
hour            byte        0
minute          byte        0
second          byte        0
Keystrokes      word        0
RecSize         =           $-month







; MyInt9-       The system calls this routine every time a keyboard
;               interrupt occus. This routine increments the
;               KeyIntCnt variable and then passes control on to the
;               original Int9 handler.

MyInt9          proc        far
                inc         ResidentSeg:KeyIntCnt
                jmp         ResidentSeg:OldInt9
MyInt9          endp




; MyInt1C-      Timer interrupt. This guy counts off 60 seconds and then
;               attempts to write a record to the output file. Of course,
;               this call has to jump through all sorts of hoops to keep
;               from reentering DOS and other problematic code.
```

```
MyInt1C         proc    far
                assume  ds:ResidentSeg

                push    ds
                push    es
                pusha                   ;Save all the registers.
                mov     ax, ResidentSeg
                mov     ds, ax

                pushf
                call    OldInt1C

; First things first, let's bump our interrupt counter so we can count
; off a minute. Since we're getting interrupted about every 54.92549
; milliseconds, let's shoot for a little more accuracy than 18 times
; per second so the timings don't drift too much.

                add     Counter, 549    ;54.9 msec per int 1C.
                cmp     Counter, 10000  ;1 second.
                jb      NotSecYet
                sub     Counter, 10000
                inc     SecCounter
NotSecYet:


; If NEEDIO is not zero, then there is an I/O operation in progress.
; Do not disturb the output values if this is the case.

                cli                     ;This is a critical region.
                cmp     NeedIO, 0
                jne     SkipSetNIO

; Okay, no I/O in progress, see if a minute has passed since the last
; time we logged the keystrokes to the file. If so, it's time to start
; another I/O operation.

                cmp     SecCounter, 60  ;One minute passed yet?
                jb      Int1CDone
                mov     NeedIO, 1       ;Flag need for I/O.
                mov     ax, KeyIntCnt   ;Copy this to the output
                shr     ax, 1           ; buffer after computing
                mov KeyStrokes, ax      ; # of keystrokes.
                mov     KeyIntCnt, 0    ;Reset for next minute.
                mov     SecCounter, 0

SkipSetNIO:     cmp     NeedIO, 1       ;Is the I/O already in
                jne     Int1CDone       ; progress? Or done?

                call    ChkDOSStatus    ;See if DOS/BIOS are free.
                jnc     Int1CDone       ;Branch if busy.

                call    DoIO            ;Do I/O if DOS is free.

Int1CDone:      popa                    ;Restore registers and quit.
                pop     es
                pop     ds
                iret
MyInt1C         endp
                assume  ds:nothing


; MyInt28-      Idle interrupt. If DOS is in a busy-wait loop waiting for
;               I/O to complete, it executes an int 28h instruction each
;               time through the loop. We can ignore the InDOS and CritErr
;               flags at that time, and do the I/O if the other interrupts
;               are free.

MyInt28         proc    far
                assume  ds:ResidentSeg

                push    ds
                push    es
                pusha                   ;Save all the registers.
```

```
                mov     ax, ResidentSeg
                mov     ds, ax

                pushf                   ;Call the next INT 28h
                call    OldInt28        ; ISR in the chain.

                cmp     NeedIO, 1       ;Do we have a pending I/O?
                jne     Int28Done

                mov     al, InInt13     ;See if BIOS is busy.
                or      al, InInt16
                jne     Int28Done

                call    DoIO            ;Go do I/O if BIOS is free.

Int28Done:      popa
                pop     es
                pop     ds
                iret
MyInt28         endp
                assume  ds:nothing


; MyInt16-      This is just a wrapper for the INT 16h (keyboard trap)
;               handler.

MyInt16         proc    far
                inc     ResidentSeg:InInt16

; Call original handler:

                pushf
                call    ResidentSeg:OldInt16

; For INT 16h we need to return the flags that come from the previous call.

                pushf
                dec     ResidentSeg:InInt16
                popf
                retf    2               ;Fake IRET to keep flags.
MyInt16         endp


; MyInt13-      This is just a wrapper for the INT 13h (disk I/O trap)
;               handler.

MyInt13         proc    far
                inc     ResidentSeg:InInt13
                pushf
                call    ResidentSeg:OldInt13
                pushf
                dec     ResidentSeg:InInt13
                popf
                retf    2               ;Fake iret to keep flags.
MyInt13         endp


; ChkDOSStatus-         Returns with the carry clear if DOS or a BIOS routine
;                       is busy and we can't interrupt them.

ChkDOSStatus    proc    near
                assume  ds:ResidentSeg
                les     bx, InDOSFlag
                mov     al, es:[bx]     ;Get InDOS flag.
                or      al, es:[bx-1]   ;OR with CritErr flag.
                or      al, InInt16     ;OR with our wrapper
                or      al, InInt13     ; values.
                je      Okay2Call
                clc
                ret

Okay2Call:      clc
                ret
ChkDOSStatus    endp
```

```
                assume  ds:nothing


; PreserveDOS-  Gets a copy's of DOS' current PSP, DTA, and extended
;               error information and saves this stuff. Then it sets
;               the PSP to our local PSP and the DTA to PSP:80h.

PreserveDOS     proc    near
                assume  ds:ResidentSeg

                mov     ah, 51h         ;Get app's PSP.
                int     21h
                mov     AppPSP, bx      ;Save for later

                mov     ah, 2Fh         ;Get app's DTA.
                int     21h
                mov     word ptr AppDTA, bx
                mov     word ptr AppDTA+2, es

                push    ds
                mov     ah, 59h         ;Get extended err info.
                xor     bx, bx
                int     21h

                mov     cs:XErr.eeDS, ds
                pop     ds
                mov     XErr.eeAX, ax
                mov     XErr.eeBX, bx
                mov     XErr.eeCX, cx
                mov     XErr.eeDX, dx
                mov     XErr.eeSI, si
                mov     XErr.eeDI, di
                mov     XErr.eeES, es

; Okay, point DOS's pointers at us:

                mov     bx, PSP
                mov     ah, 50h         ;Set PSP.
                int     21h

                push    ds              ;Set the DTA to
                mov     ds, PSP         ; address PSP:80h
                mov     dx, 80h
                mov     ah, 1Ah         ;Set DTA call.
                int     21h
                pop     ds

                ret
PreserveDOS     endp
                assume  ds:nothing


; RestoreDOS-   Restores DOS' important global data values back to the
;               application's values.

RestoreDOS      proc    near
                assume  ds:ResidentSeg

                mov     bx, AppPSP
                mov     ah, 50h         ;Set PSP
                int     21h

                push    ds
                lds     dx, AppDTA
                mov     ah, 1Ah         ;Set DTA
                int     21h
                pop     ds
                push    ds

                mov     si, offset XErr ;Saved extended error stuff.
                mov     ax, 5D0Ah       ;Restore XErr call.
                int     21h
                pop     ds
```

```
                ret
RestoreDOS      endp
                assume  ds:nothing


; DoIO-         This routine processes each of the I/O operations
;               required to write data to the file.

DoIO            proc    near
                assume  ds:ResidentSeg

                mov     NeedIO, 0FFh    ;A busy flag for us.

; The following Get Date DOS call may take a while, so turn the
; interrupts back on (we're clear of the critical section once we
; write 0FFh to NeedIO).

                sti
                call    PreserveDOS     ;Save DOS data.

                mov     ah, 2Ah         ;Get Date DOS call
                int     21h
                mov     month, dh
                mov     day, dl
                mov     year, cx

                mov     ah, 2Ch         ;Get Time DOS call
                int     21h
                mov     hour, ch
                mov     minute, cl
                mov     second, dh

                mov     ah, 40h         ;DOS Write call
                mov     bx, FileHandle  ;Write data to this file.
                mov     cx, RecSize     ;This many bytes.
                mov     dx, offset month ;Starting at this address.
                int     21h             ;Ignore return errors (!).
                mov     ah, 68h         ;DOS Commit call
                mov     bx, FileHandle  ;Write data to this file.
                int     21h             ;Ignore return errors (!).

                mov     NeedIO, 0       ;Ready to start over.
                call    RestoreDOS

PhasesDone:     ret
DoIO            endp
                assume  ds:nothing



; MyInt2F-      Provides int 2Fh (multiplex interrupt) support for this
;               TSR. The multiplex interrupt recognizes the following
;               subfunctions (passed in AL):
;
;               00- Verify presence.    Returns 0FFh in AL and a pointer
;                                       to an ID string in es:di if the
;                                       TSR ID (in AH) matches this
;                                       particular TSR.
;
;               01- Remove.             Removes the TSR from memory.
;                                       Returns 0 in AL if successful,
;                                       1 in AL if failure.

MyInt2F         proc    far
                assume  ds:nothing

                cmp     ah, MyTSRID     ;Match our TSR identifier?
                je      YepItsOurs
                jmp     OldInt2F

; Okay, we know this is our ID, now check for a verify vs. remove call.

YepItsOurs:     cmp     al, 0           ;Verify Call
                jne     TryRmv
```

```
                mov     al, 0ffh        ;Return success.
                lesi    IDString
                iret                    ;Return back to caller.

IDString        byte    "Keypress Logger TSR",0

TryRmv:         cmp     al, 1           ;Remove call.
                jne     IllegalOp

                call    TstRmvable      ;See if we can remove this guy.
                je      CanRemove       ;Branch if we can.
                mov     ax, 1           ;Return failure for now.
                iret

; Okay, they want to remove this guy *and* we can remove it from memory.
; Take care of all that here.

                assume  ds:ResidentSeg

CanRemove:      push    ds
                push    es
                pusha
                cli                     ;Turn off the interrupts while
                mov     ax, 0           ; we mess with the interrupt
                mov     es, ax          ; vectors.
                mov     ax, cs
                mov     ds, ax

                mov     ax, word ptr OldInt9
                mov     es:[9*4], ax
                mov     ax, word ptr OldInt9+2
                mov     es:[9*4 + 2], ax

                mov     ax, word ptr OldInt13
                mov     es:[13h*4], ax
                mov     ax, word ptr OldInt13+2
                mov     es:[13h*4 + 2], ax

                mov     ax, word ptr OldInt16
                mov     es:[16h*4], ax
                mov     ax, word ptr OldInt16+2
                mov     es:[16h*4 + 2], ax

                mov     ax, word ptr OldInt1C
                mov     es:[1Ch*4], ax
                mov     ax, word ptr OldInt1C+2
                mov     es:[1Ch*4 + 2], ax

                mov     ax, word ptr OldInt28
                mov     es:[28h*4], ax
                mov     ax, word ptr OldInt28+2
                mov     es:[28h*4 + 2], ax

                mov     ax, word ptr OldInt2F
                mov     es:[2Fh*4], ax
                mov     ax, word ptr OldInt2F+2
                mov     es:[2Fh*4 + 2], ax


; Okay, with that out of the way, let's close the file.
; Note: INT 2F shouldn't have to deal with DOS busy because it's
; a passive TSR call.

                mov     ah, 3Eh         ;Close file command
                mov     bx, FileHandle
                int     21h

; Okay, one last thing before we quit- Let's give the memory allocated
; to this TSR back to DOS.

                mov     ds, PSP
                mov     es, ds:[2Ch]    ;Ptr to environment block.
                mov     ah, 49h         ;DOS release memory call.
                int     21h
```

```
                mov     ax, ds          ;Release program code space.
                mov     es, ax
                mov     ah, 49h
                int     21h

                popa
                pop     es
                pop     ds
                mov     ax, 0           ;Return Success.
                iret


; They called us with an illegal subfunction value. Try to do as little
; damage as possible.

IllegalOp:      mov     ax, 0           ;Who knows what they were thinking?
                iret
MyInt2F         endp
                assume  ds:nothing




; TstRmvable-   Checks to see if we can remove this TSR from memory.
;               Returns the zero flag set if we can remove it, clear
;               otherwise.

TstRmvable      proc    near
                cli
                push    ds
                mov     ax, 0
                mov     ds, ax

                cmp     word ptr ds:[9*4], offset MyInt9
                jne     TRDone
                cmp     word ptr ds:[9*4 + 2], seg MyInt9
                jne     TRDone

                cmp     word ptr ds:[13h*4], offset MyInt13
                jne     TRDone
                cmp     word ptr ds:[13h*4 + 2], seg MyInt13
                jne     TRDone

                cmp     word ptr ds:[16h*4], offset MyInt16
                jne     TRDone
                cmp     word ptr ds:[16h*4 + 2], seg MyInt16
                jne     TRDone

                cmp     word ptr ds:[1Ch*4], offset MyInt1C
                jne     TRDone
                cmp     word ptr ds:[1Ch*4 + 2], seg MyInt1C
                jne     TRDone

                cmp     word ptr ds:[28h*4], offset MyInt28
                jne     TRDone
                cmp     word ptr ds:[28h*4 + 2], seg MyInt28
                jne     TRDone

                cmp     word ptr ds:[2Fh*4], offset MyInt2F
                jne     TRDone
                cmp     word ptr ds:[2Fh*4 + 2], seg MyInt2F
TRDone:         pop     ds
                sti
                ret
TstRmvable      endp
ResidentSeg     ends



cseg            segment para public 'code'
                assume  cs:cseg, ds:ResidentSeg
```

```
; SeeIfPresent-         Checks to see if our TSR is already present in memory.
;                       Sets the zero flag if it is, clears the zero flag if
;                       it is not.

SeeIfPresent  proc      near
              push      es
              push      ds
              push      di
              mov       cx, 0ffh          ;Start with ID 0FFh.
IDLoop:       mov       ah, cl
              push      cx
              mov       al, 0             ;Verify presence call.
              int       2Fh
              pop       cx
              cmp       al, 0             ;Present in memory?
              je        TryNext
              strcmpl
              byte      "Keypress Logger TSR",0
              je        Success

TryNext:      dec       cl                ;Test USER IDs of 80h..FFh
              js        IDLoop
              cmp       cx, 0             ;Clear zero flag.
Success:      pop       di
              pop       ds
              pop       es
              ret
SeeIfPresent  endp


; FindID-     Determines the first (well, last actually) TSR ID available
;             in the multiplex interrupt chain. Returns this value in
;             the CL register.
;
;             Returns the zero flag set if it locates an empty slot.
;             Returns the zero flag clear if failure.

FindID        proc      near
              push      es
              push      ds
              push      di

              mov       cx, 0ffh          ;Start with ID 0FFh.
IDLoop:       mov       ah, cl
              push      cx
              mov       al, 0             ;Verify presence call.
              int       2Fh
              pop       cx
              cmp       al, 0             ;Present in memory?
              je        Success
              dec       cl                ;Test USER IDs of 80h..FFh
              js        IDLoop
              xor       cx, cx
              cmp       cx, 1             ;Clear zero flag
Success:      pop       di
              pop       ds
              pop       es
              ret
FindID        endp


Main          proc
              meminit

              mov       ax, ResidentSeg
              mov       ds, ax

              mov       ah, 62h           ;Get this program's PSP
              int       21h               ; value.
              mov       PSP, bx

; Before we do anything else, we need to check the command line
```

```
; parameters. We must have either a valid filename or the
; command "remove". If remove appears on the command line, then remove
; the resident copy from memory using the multiplex (2Fh) interrupt.
; If remove is not on the command line, we'd better have a filename and
; there had better not be a copy already loaded into memory.

                argc
                cmp         cx, 1             ;Must have exactly 1 parm.
                je          GoodParmCnt
                print
                byte        "Usage:",cr,lf
                byte        " KeyEval filename",cr,lf
                byte        "or KeyEval REMOVE",cr,lf,0
                ExitPgm


; Check for the REMOVE command.

GoodParmCnt:    mov         ax, 1
                argv
                stricmpl
                byte        "REMOVE",0
                jne         TstPresent

                call        SeeIfPresent
                je          RemoveIt
                print
                byte        "TSR is not present in memory, cannot remove"
                byte        cr,lf,0
                ExitPgm

RemoveIt:       mov         MyTSRID, cl
                printf
                byte        "Removing TSR (ID #%d) from memory...",0
                dword       MyTSRID

                mov         ah, cl
                mov         al, 1             ;Remove cmd, ah contains ID
                int         2Fh
                cmp         al, 1             ;Succeed?
                je          RmvFailure
                print
                byte        "removed.",cr,lf,0
                ExitPgm

RmvFailure:     print
                byte        cr,lf
                byte        "Could not remove TSR from memory.",cr,lf
                byte        "Try removing other TSRs in the reverse order "
                byte        "you installed them.",cr,lf,0
                ExitPgm


; Okay, see if the TSR is already in memory. If so, abort the
; installation process.

TstPresent:     call        SeeIfPresent
                jne         GetTSRID
                print
                byte        "TSR is already present in memory.",cr,lf
                byte        "Aborting installation process",cr,lf,0
                ExitPgm


; Get an ID for our TSR and save it away.

GetTSRID:       call        FindID
                je          GetFileName
                print
                byte        "Too many resident TSRs, cannot install",cr,lf,0
                ExitPgm
```

```
; Things look cool so far, check the filename and open the file.

GetFileName:    mov         MyTSRID, cl
                printf
                byte        "Keypress logger TSR program",cr,lf
                byte        "TSR ID = %d",cr,lf
                byte        "Processing file:",0
                dword       MyTSRID

                puts
                putcr

                mov         ah, 3Ch             ;Create file command.
                mov         cx, 0               ;Normal file.
                push        ds
                push        es                  ;Point ds:dx at name
                pop         ds
                mov         dx, di
                int         21h                 ;Open the file
                jnc         GoodOpen
                print
                byte        "DOS error #",0
                puti
                print
                byte        " opening file.",cr,lf,0
                ExitPgm

GoodOpen:       pop         ds
                mov         FileHandle, ax      ;Save file handle.


InstallInts:    print
                byte        "Installing interrupts...",0


; Patch into the INT 9, 13h, 16h, 1Ch, 28h, and 2Fh interrupt vectors.
; Note that the statements above have made ResidentSeg the current data
; segment, so we can store the old values directly into
; the OldIntxx variables.

                cli                             ;Turn off interrupts!
                mov         ax, 0
                mov         es, ax
                mov         ax, es:[9*4]
                mov         word ptr OldInt9, ax
                mov         ax, es:[9*4 + 2]
                mov         word ptr OldInt9+2, ax
                mov         es:[9*4], offset MyInt9
                mov         es:[9*4+2], seg ResidentSeg

                mov         ax, es:[13h*4]
                mov         word ptr OldInt13, ax
                mov         ax, es:[13h*4 + 2]
                mov         word ptr OldInt13+2, ax
                mov         es:[13h*4], offset MyInt13
                mov         es:[13h*4+2], seg ResidentSeg

                mov         ax, es:[16h*4]
                mov         word ptr OldInt16, ax
                mov         ax, es:[16h*4 + 2]
                mov         word ptr OldInt16+2, ax
                mov         es:[16h*4], offset MyInt16
                mov         es:[16h*4+2], seg ResidentSeg

                mov         ax, es:[1Ch*4]
                mov         word ptr OldInt1C, ax
                mov         ax, es:[1Ch*4 + 2]
                mov         word ptr OldInt1C+2, ax
                mov         es:[1Ch*4], offset MyInt1C
                mov         es:[1Ch*4+2], seg ResidentSeg

                mov         ax, es:[28h*4]
                mov         word ptr OldInt28, ax
                mov         ax, es:[28h*4 + 2]
```

```
                mov     word ptr OldInt28+2, ax
                mov     es:[28h*4], offset MyInt28
                mov     es:[28h*4+2], seg ResidentSeg

                mov     ax, es:[2Fh*4]
                mov     word ptr OldInt2F, ax
                mov     ax, es:[2Fh*4 + 2]
                mov     word ptr OldInt2F+2, ax
                mov     es:[2Fh*4], offset MyInt2F
                mov     es:[2Fh*4+2], seg ResidentSeg
                sti                     ;Okay, ints back on.

; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte    "Installed.",cr,lf,0


                mov     dx, EndResident ;Compute size of program.
                sub     dx, PSP
                mov     ax, 3100h       ;DOS TSR command.
                int     21h
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

The following is a short little application that reads the data file produced by the above program and produces a simple report of the date, time, and keystrokes:

```
; This program reads the file created by the KEYEVAL.EXE TSR program.
; It displays the log containing dates, times, and number of keystrokes.

                .xlist
                .286
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'

FileHandle      word       ?

month           byte       0
day             byte       0
year            word       0
hour            byte       0
minute          byte       0
second          byte       0
KeyStrokes      word       0
RecSize         =          $-month

dseg            ends



cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg
```

```
; SeeIfPresent-         Checks to see if our TSR is present in memory.
;                       Sets the zero flag if it is, clears the zero flag if
;                       it is not.

SeeIfPresent  proc      near
              push      es
              push      ds
              pusha
              mov       cx, 0ffh          ;Start with ID 0FFh.
IDLoop:       mov       ah, cl
              push      cx
              mov       al, 0             ;Verify presence call.
              int       2Fh
              pop       cx
              cmp       al, 0             ;Present in memory?
              je        TryNext
              strcmpl
              byte      "Keypress Logger TSR",0
              je        Success

TryNext:      dec       cl                ;Test USER IDs of 80h..FFh
              js        IDLoop
              cmp       cx, 0             ;Clear zero flag.
Success:      popa
              pop       ds
              pop       es
              ret
SeeIfPresent  endp



Main          proc
              meminit

              mov       ax, dseg
              mov       ds, ax



              argc
              cmp       cx, 1             ;Must have exactly 1 parm.
              je        GoodParmCnt
              print
              byte      "Usage:",cr,lf
              byte      " KEYRPT filename",cr,lf,0
              ExitPgm



GoodParmCnt:  mov       ax, 1
              argv

              print
              byte      "Keypress logger report program",cr,lf
              byte      "Processing file:",0
              puts
              putcr

              mov       ah, 3Dh           ;Open file command.
              mov       al, 0             ;Open for reading.
              push      ds
              push      es                ;Point ds:dx at name
              pop       ds
              mov       dx, di
              int       21h               ;Open the file
              jnc       GoodOpen
              print
              byte      "DOS error #",0
              puti
              print
              byte      " opening file.",cr,lf,0
              ExitPgm
```

```
GoodOpen:       pop         ds
                mov         FileHandle, ax      ;Save file handle.


; Okay, read the data and display it:

ReadLoop:       mov         ah, 3Fh             ;Read file command
                mov         bx, FileHandle
                mov         cx, RecSize         ;Number of bytes.
                mov         dx, offset month    ;Place to put data.
                int         21h
                jc          ReadError
                test        ax, ax              ;EOF?
                je          Quit

                mov         cx, year
                mov         dl, day
                mov         dh, month
                dtoam
                puts
                free
                print
                byte        ", ",0

                mov         ch, hour
                mov         cl, minute
                mov         dh, second
                mov         dl, 0
                ttoam
                puts
                free
                printf
                byte        ", keystrokes = %d\n",0
                dword       KeyStrokes
                jmp         ReadLoop

ReadError:      print
                byte        "Error reading file",cr,lf,0

Quit:           mov         bx, FileHandle
                mov         ah, 3Eh             ;Close file
                int         21h
                ExitPgm

Main            endp
cseg            ends

sseg            segment     para stack 'stack'
stk             db          1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

18.9 Semiresident Programs

A *semiresident* program is one that temporarily loads itself into memory, executes another program (a child process), and then removes itself from memory after the child process terminates. Semiresident programs behave like resident programs while the child executes, but they do not stay in memory once the child terminates.

The main use for semiresident programs is to extend an existing application or *patch* an application[6] (the child process). The nice thing about a semiresident program patch is that it does not have to modify

6. *Patching* a program means to replace certain opcode bytes in the object file. Programmers apply patches to correct bugs or extend a product whose sources are not available.

the application's ".EXE" file directly on the disk. If for some reason the patch fails, you haven't destroyed the '.EXE" file, you've only wiped out the object code in memory.

A semiresident application, like a TSR, has a transient and a resident part. The resident part remains in memory while the child process executes. The transient part initializes the program and then transfers control to the resident part that loads the child application over the resident portion. The transient code patches the interrupt vectors and does all the things a TSR does *except it doesn't issue the TSR command.* Instead, the resident program loads the application into memory and transfers control to that program. When the application returns control to the resident program, it exits to DOS using the standard ExitPgm call (ah=4Ch).

While the application is running, the resident code behaves like any other TSR. Unless the child process is aware of the semiresident program, or the semiresident program patches interrupt vectors the application normally uses, the semiresident program will probably be an active resident program, patching into one or more of the hardware interrupts. Of course, all the rules that apply to active TSRs also apply to active semiresident programs.

The following is a very generic example of s semiresident program. This program, "RUN.ASM", runs the application whose name and command line parameters appear as command line parameters to run. In other words:

```
c:> run pgm.exe parm1 parm2 etc.
```

is equivalent to

```
pgm parm1 parm2 etc.
```

Note that you must supply the ".EXE" or ".COM" extension to the program's filename. This code begins by extracting the program's filename and command line parameters from run's command line. Run builds an exec structure (see "MS-DOS, PC-BIOS, and File I/O" on page 699) and then calls DOS to execute the program. On return, run fixes up the stack and returns to DOS.

```
; RUN.ASM - The barebones semiresident program.
;
;       Usage:
;             RUN <program.exe> <program's command line>
;         or  RUN <program.com> <program's command line>
;
; RUN executes the specified program with the supplied command line parameters.
; At first, this may seem like a stupid program. After all, why not just run
; the program directly from DOS and skip the RUN altogether? Actually, there
; is a good reason for RUN-- It lets you (by modifying the RUN source file)
; set up some environment prior to running the program and clean up that
; environment after the program terminates ("environment" in this sense does
; not necessarily refer to the MS-DOS ENVIRONMENT area).
;
; For example, I have used this program to switch the mode of a TSR prior to
; executing an EXE file and then I restored the operating mode of that TSR
; after the program terminated.
;
; In general, you should create a new version of RUN.EXE (and, presumbably,
; give it a unique name) for each application you want to use this program
; with.
;
;
;-----------------------------------------------------------------------------
;
;
; Put these segment definitions 1st because we want the Standard Library
; routines to load last in memory, so they wind up in the transient portion.

CSEG            segment    para public 'CODE'
CSEG            ends
SSEG            segment    para stack 'stack'
SSEG            ends
ZZZZZZSEG       segment    para public 'zzzzzzseg'
ZZZZZZSEG       ends
```

```
; Includes for UCR Standard Library macros.

                include    consts.a
                include stdin.a
                include stdout.a
                include misc.a
                include memory.a
                include    strings.a

                includelib stdlib.lib


CSEG            segment    para public 'CODE'
                assume     cs:cseg, ds:cseg


; Variables used by this program.


; MS-DOS EXEC structure.

ExecStruct      dw         0                ;Use parent's Environment blk.
                dd         CmdLine          ;For the cmd ln parms.
                dd         DfltFCB
                dd         DfltFCB

DfltFCB         db         3," ",0,0,0,0,0
CmdLine         db         0, 0dh, 126 dup (" ") ;Cmd line for program.
PgmName         dd         ?                ;Points at pgm name.




Main            proc
                mov        ax, cseg         ;Get ptr to vars segment
                mov        ds, ax

                MemInit                     ;Start the memory mgr.


; If you want to do something before the execution of the command-line
; specified program, here is a good place to do it:


;       ------------------------------------


; Now let's fetch the program name, etc., from the command line and execute
; it.

                argc                        ;See how many cmd ln parms
                or         cx, cx           ; we have.
                jz         Quit             ;Just quit if no parameters.

                mov        ax, 1            ;Get the first parm (pgm name)
                argv
                mov        word ptr PgmName, di;Save ptr to name
                mov        word ptr PgmName+2, es


; Okay, for each word on the command line after the filename, copy
; that word to CmdLine buffer and separate each word with a space,
; just like COMMAND.COM does with command line parameters it processes.

                lea        si, CmdLine+1  ;Index into cmdline.
ParmLoop:       dec        cx
                jz         ExecutePgm

                inc        ax               ;Point at next parm.
                argv                        ;Get the next parm.
```

```
                push    ax
                mov     byte ptr [si], ' ' ;1st item and separator on ln.
                inc     CmdLine
                inc     si
CpyLp:          mov     al, es:[di]
                cmp     al, 0
                je      StrDone
                inc     CmdLine         ;Increment byte cnt
                mov     ds:[si], al
                inc     si
                inc     di
                jmp     CpyLp

StrDone:        mov     byte ptr ds:[si], cr ;In case this is the end.
                pop     ax              ;Get current parm #
                jmp     ParmLoop


; Okay, we've built the MS-DOS execute structure and the necessary
; command line, now let's see about running the program.
; The first step is to free up all the memory that this program
; isn't using. That would be everything from zzzzzzseg on.

ExecutePgm:     mov     ah, 62h         ;Get our PSP value
                int     21h
                mov     es, bx
                mov     ax, zzzzzzseg   ;Compute size of
                sub     ax, bx          ; resident run code.
                mov     bx, ax
                mov     ah, 4ah         ;Release unused memory.
                int     21h

; Warning! No Standard Library calls after this point. We've just
; released the memory that they're sitting in. So the program load
; we're about to do will wipe out the Standard Library code.

                mov     bx, seg ExecStruct
                mov     es, bx
                mov     bx, offset ExecStruct ;Ptr to program record.
                lds     dx, PgmName
                mov     ax, 4b00h       ;Exec pgm
                int     21h

; When we get back, we can't count on *anything* being correct. First, fix
; the stack pointer and then we can finish up anything else that needs to
; be done.

                mov     ax, sseg
                mov     ss, ax
                mov     sp, offset EndStk
                mov     ax, seg cseg
                mov     ds, ax

; Okay, if you have any great deeds to do after the program, this is a
; good place to put such stuff.

;       ------------------------------------

; Return control to MS-DOS

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment   para stack 'stack'
                dw        128 dup (0)
endstk          dw        ?
sseg            ends

; Set aside some room for the heap.

zzzzzzseg       segment   para public 'zzzzzzseg'
Heap            db        200h dup (?)
```

```
zzzzzzseg       ends

                end         Main
```

Since RUN.ASM is rather simple perhaps a more complex example is in order. The following is a fully functional patch for the Lucasart's game XWING™. The motivation for this patch can about because of the annoyance of having to look up a password everytime you play the game. This little patch searches for the code that calls the password routine and stores NOPs over that code in memory.

The operation of this code is a little different than that of RUN.ASM. The RUN program sends an execute command to DOS that runs the desired program. All system changes RUN needs to make must be made before or after the application executes. XWPATCH operates a little differently. It loads the XWING.EXE program into memory and searches for some specific code (the call to the password routine). Once it finds this code, it stores NOP instructions over the top of the call.

Unfortunately, life isn't quite that simple. When XWING.EXE loads, the password code isn't yet present in memory. XWING loads that code as an overlay later on. So the XWPATCH program finds something that XWING.EXE does load into memory right away – the joystick code. XWPATCH patches the joystick code so that any call to the joystick routine (when detecting or calibrating the joystick) produces a call to XWPATCH's code that searches for the password code. Once XWPATCH locates and NOPs out the call to the password routine, it restores the code in the joystick routine. From that point forward, XWPATCH is simply taking up memory space; XWING will never call it again until XWING terminates.

```
; XWPATCH.ASM
;
;       Usage:
;             XWPATCH    - must be in same directory as XWING.EXE
;
; This program executes the XWING.EXE program and patches it to avoid
; having to enter the password every time you run it.
;
; This program is intended for educational purposes only.
; It is a demonstration of how to write a semiresident program.
; It is not intended as a device to allow the piracy of commercial software.
; Such use is illegal and is punishable by law.
;
; This software is offered without warranty or any expectation of
; correctness. Due to the dynamic nature of software design, programs
; that patch other programs may not work with slight changes in the
; patched program (XWING.EXE). USE THIS CODE AT YOUR OWN RISK.
;
;-----------------------------------------------------------------------------


byp           textequ    <byte ptr>
wp            textequ    <word ptr>

; Put these segment definitions here so the UCR Standard Library will
; load after zzzzzzseg (in the transient section).

cseg          segment para public 'CODE'
cseg          ends

sseg          segment    para stack 'STACK'
sseg          ends

zzzzzzseg     segment    para public 'zzzzzzseg'
zzzzzzseg     ends

              .286
              include       stdlib.a
              includelib stdlib.lib


CSEG          segment    para public 'CODE'
```

```
                assume  cs:cseg, ds:nothing


; CountJSCalls-Number of times xwing calls the Joystick code before
; we patch out the password call.

CountJSCalls    dw      250


; PSP-  Program Segment Prefix. Needed to free up memory before running
;       the real application program.

PSP             dw      0



; Program Loading data structures (for DOS).

ExecStruct      dw      0               ;Use parent's Environment blk.
                dd      CmdLine         ;For the cmd ln parms.
                dd      DfltFCB
                dd      DfltFCB
LoadSSSP        dd      ?
LoadCSIP        dd      ?
PgmName         dd      Pgm

DfltFCB         db      3," ",0,0,0,0,0
CmdLine         db      2, " ", 0dh, 16 dup (" ");Cmd line for program
Pgm             db      "XWING.EXE",0


;*****************************************************************************
;
; XWPATCH begins here. This is the memory resident part. Only put code
; which which has to be present at run-time or needs to be resident after
; freeing up memory.
;*****************************************************************************
;

Main            proc
                mov     cs:PSP, ds
                mov     ax, cseg        ;Get ptr to vars segment
                mov     ds, ax

                mov     ax, zzzzzzseg
                mov     es, ax
                mov     cx, 1024/16
                meminit2


; Now, free up memory from ZZZZZZSEG on to make room for XWING.
; Note: Absolutely no calls to UCR Standard Library routines from
; this point forward! (ExitPgm is okay, it's just a macro which calls DOS.)
; Note that after the execution of this code, none of the code & data
; from zzzzzzseg on is valid.

                mov     bx, zzzzzzseg
                sub     bx, PSP
                inc     bx
                mov     es, PSP
                mov     ah, 4ah
                int     21h
                jnc     GoodRealloc

; Okay, I lied. Here's a StdLib call, but it's okay because we failed
; to load the application over the top of the standard library code.
; But from this point on, absolutely no more calls!

                print
                byte    "Memory allocation error."
                byte    cr,lf,0
                jmp     Quit

GoodRealloc:

; Now load the XWING program into memory:
```

```
                mov     bx, seg ExecStruct
                mov     es, bx
                mov     bx, offset ExecStruct ;Ptr to program record.
                lds     dx, PgmName
                mov     ax, 4b01h       ;Load, do not exec, pgm
                int     21h
                jc      Quit            ;If error loading file.

; Unfortunately, the password code gets loaded dynamically later on.
; So it's not anywhere in memory where we can search for it. But we
; do know that the joystick code is in memory, so we'll search for
; that code. Once we find it, we'll patch it so it calls our SearchPW
; routine. Note that you must use a joystick (and have one installed)
; for this patch to work properly.

                mov     si, zzzzzzseg
                mov     ds, si
                xor     si, si

                mov     di, cs
                mov     es, di
                mov     di, offset JoyStickCode
                mov     cx, JoyLength
                call    FindCode
                jc      Quit            ;If didn't find joystick code.


; Patch the XWING joystick code here

                mov     byp ds:[si], 09ah;Far call
                mov     wp ds:[si+1], offset SearchPW
                mov     wp ds:[si+3], cs

; Okay, start the XWING.EXE program running

                mov     ah, 62h         ;Get PSP
                int     21h
                mov     ds, bx
                mov     es, bx
                mov     wp ds:[10], offset Quit
                mov     wp ds:[12], cs
                mov     ss, wp cseg:LoadSSSP+2
                mov     sp, wp cseg:LoadSSSP
                jmp     dword ptr cseg:LoadCSIP


Quit:           ExitPgm
Main            endp


; SearchPW gets call from XWING when it attempts to calibrate the joystick.
; We'll let XWING call the joystick several hundred times before we
; actually search for the password code. The reason we do this is because
; XWING calls the joystick code early on to test for the presence of a
; joystick. Once we get into the calibration code, however, it calls
; the joystick code repetitively, so a few hundred calls doesn't take
; very long to expire. Once we're in the calibration code, the password
; code has been loaded into memory, so we can search for it then.

SearchPW        proc    far
                cmp     cs:CountJSCalls, 0
                je      DoSearch
                dec     cs:CountJSCalls
                sti                     ;Code we stole from xwing for
                neg     bx              ; the patch.
                neg     di
                ret

; Okay, search for the password code.

DoSearch:       push    bp
                mov     bp, sp
                push    ds
```

```
                push    es
                pusha

; Search for the password code in memory:

                mov     si, zzzzzzseg
                mov     ds, si
                xor     si, si

                mov     di, cs
                mov     es, di
                mov     di, offset PasswordCode
                mov     cx, PWLength
                call    FindCode
                jc      NotThere        ;If didn't find pw code.


; Patch the XWING password code here. Just store NOPs over the five
; bytes of the far call to the password routine.

                mov     byp ds:[si+11], 090h            ;NOP out a far call
                mov     byp ds:[si+12], 090h
                mov     byp ds:[si+13], 090h
                mov     byp ds:[si+14], 090h
                mov     byp ds:[si+15], 090h

; Adjust the return address and restore the patched joystick code so
; that it doesn't bother jumping to us anymore.

NotThere:       sub     word ptr [bp+2], 5 ;Back up return address.
                les     bx, [bp+2]      ;Fetch return address.

; Store the original joystick code over the call we patched to this
; routine.

                mov     ax, word ptr JoyStickCode
                mov     es:[bx], ax
                mov     ax, word ptr JoyStickCode+2
                mov     es:[bx+2], ax
                mov     al, byte ptr JoyStickCode+4
                mov     es:[bx+4], al

                popa
                pop     es
                pop     ds
                pop     bp
                ret
SearchPW        endp

;*****************************************************************************
;
; FindCode: On entry, ES:DI points at some code in *this* program which
;         appears in the XWING game. DS:SI points at a block of memory
;         in the XWING game. FindCode searches through memory to find the
;         suspect piece of code and returns DS:SI pointing at the start of
;         that code. This code assumes that it *will* find the code!
;         It returns the carry clear if it finds it, set if it doesn't.

FindCode        proc    near
                push    ax
                push    bx
                push    dx

DoCmp:          mov     dx, 1000h       ;Search in 4K blocks.
CmpLoop:        push    di              ;Save ptr to compare code.
                push    si              ;Save ptr to start of string.
                push    cx              ;Save count.
          repe  cmpsb
                pop     cx
                pop     si
                pop     di
                je      FoundCode
                inc     si
                dec     dx
```

```
                jne         CmpLoop
                sub         si, 1000h
                mov         ax, ds
                inc         ah
                mov         ds, ax
                cmp         ax, 9000h           ;Stop at address 9000:0
                jb          DoCmp               ; and fail if not found.

                pop         dx
                pop         bx
                pop         ax
                stc
                ret

FoundCode:      pop         dx
                pop         bx
                pop         ax
                clc
                ret
FindCode        endp


;*****************************************************************************
;
; Call to password code that appears in the XWING game. This is actually
; data that we're going to search for in the XWING object code.

PasswordCode    proc        near
                call        $+47h
                mov         [bp-4], ax
                mov         [bp-2], dx
                push        dx
                push        ax
                byte        9ah, 04h, 00
PasswordCode    endp
EndPW:

PWLength        =           EndPW-PasswordCode


; The following is the joystick code we're going to search for.

JoyStickCode    proc        near
                sti
                neg         bx
                neg         di
                pop         bp
                pop         dx
                pop         cx
                ret
                mov         bp, bx
                in          al, dx
                mov         bl, al
                not         al
                and         al, ah
                jnz         $+11h
                in          al, dx
JoyStickCode    endp
EndJSC:

JoyLength       =           EndJSC-JoyStickCode
cseg            ends

sseg            segment     para stack 'STACK'
                dw          256 dup (0)
endstk          dw          ?
sseg            ends

zzzzzzseg       segment     para public 'zzzzzzseg'
Heap            db          1024 dup (0)
zzzzzzseg       ends
                end         Main
```

18.10 Summary

Resident programs provide a small amount of multitasking to DOS' single tasking world. DOS provides support for resident programs through a rudimentary memory management system. When an application issues the terminate and stay resident call, DOS adjusts its memory pointers so the memory space reserved by the TSR code is protected from future program loading operations. For more information on how this process works, see

- "DOS Memory Usage and TSRs" on page 1025

TSRs come in two basic forms: active and passive. Passive TSRs are not self-activating. A foreground application must call a routine in a passive TSR to activate it. Generally, an application interfaces to a passive TSR using the 80x86 trap mechanism (software interrupts). Active TSRs, on the other hand, do not rely on the foreground application for activation. Instead, they attach themselves to a hardware interrupt that activates them independently of the foreground process. For more information, see

- "Active vs. Passive TSRs" on page 1029

The nature of an active TSR introduces many compatibility problems. The primary problem is that an active TSR might want to call a DOS or BIOS routine after having just interrupted either of these systems. This creates problems because DOS and BIOS are not *reentrant*. Fortunately, MS-DOS provides some hooks that give active TSRs the ability to schedule DOS calls with DOS is inactive. Although the BIOS routines do not provide this same facility, it is easy to add a *wrapper* around a BIOS call to let you schedule calls appropriately. One additional problem with DOS is that an active TSR might disturb some global variable in use by the foreground process. Fortunately, DOS lets the TSR save and restore these values, preventing some nasty compatibility problems. For details, see

- "Reentrancy" on page 1032
- "Reentrancy Problems with DOS" on page 1032
- "Reentrancy Problems with BIOS" on page 1033
- "Reentrancy Problems with Other Code" on page 1034
- "Other DOS Related Issues" on page 1039

MS-DOS provides a special interrupt to coordinate communication between TSRs and other applications. The *multiplex* interrupt lets you easily check for the presence of a TSR in memory, remove a TSR from memory, or pass various information between the TSR and an active application. For more information, see

- "The Multiplex Interrupt (INT 2Fh)" on page 1034

Well written TSRs follow stringent rules. In particular, a good TSR follows certain conventions during installation and always provide the user with a safe removal mechanism that frees all memory in use by the TSR. In those rare cases where a TSR cannot remove itself, it always reports an appropriate error and instructs the user how to solve the problem. For more information on load and removing TSRs, see

- "Installing a TSR" on page 1035
- "Removing a TSR" on page 1037
- "A Keyboard Monitor TSR" on page 1041

A semiresident routine is one that is resident during the execution of some specific program. It automatically unloads itself when that application terminates. Semiresident applications find application as program patchers and "time-release TSRs." For more information on semiresident programs, see

- "Semiresident Programs" on page 1055

Processes, Coroutines, and Concurrency Chapter 19

When most people speak of multitasking, they usually mean the ability to run several different application programs concurrently on one machine. Given the structure of the original 80x86 chips and MS-DOS' software design, this is very difficult to achieve when running DOS. Look at how long it's taken Microsoft to get Windows to multitask as well as it does.

Given the problems large companies like Microsoft have had trying to get multitasking to work, you might thing that it is a very difficult thing to manage. However, this isn't true. Microsoft has problems trying to make different applications *that are unaware of one another* work harmoniously together. Quite frankly, they have not succeeded in getting existing DOS applications to multitask well. Instead, they've been working on developers to write new programs that work well under Windows.

Multitasking is not trivial, but it is not that difficult when you write an application with multitasking specifically in mind. You can even write programs that multitask under DOS if you only take a few precautions. In this chapter, we will discuss the concept of a DOS *process*, a *coroutine*, and a general *process*.

19.1 DOS Processes

Although MS-DOS is a single tasking operating system, this does not mean there can only be one program at a time in memory. Indeed, the whole purpose of the previous chapter was to describe how to get two or more programs operating in memory at one time. However, even if we ignore TSRs for the time being, you can still load several programs into memory at one time under DOS. The only catch is, DOS only provides the ability for them to run one at a time in a very specific fashion. Unless the processes are *cooperating*, their execution profile follows a very strict pattern.

19.1.1 Child Processes in DOS

When a DOS application is running, it can load and executing some other program using the DOS EXEC function (see "MS-DOS, PC-BIOS, and File I/O" on page 699). Under normal circumstances, when an application (the parent) runs a second program (the child), the child process executes to completion and then returns to the parent. This is very much like a procedure call, except it is a little more difficult to pass parameters between the two.

MS-DOS provides several functions you can use to load and execute program code, terminate processes, and obtain the exit status for a process. The following table lists many of these operations.

Table 67: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
4Bh	al- 0 ds:dx- pointer to program name. es:bx- pointer to LOADEXEC structure.	ax- error code if carry set.	Load and execute program
4Bh	al- 1 ds:dx- pointer to program name. es:bx- pointer to LOAD structure.	ax- error code if carry set.	Load program
4Bh	al- 3 ds:dx- pointer to program name. es:bx- pointer to OVERLAY structure.	ax- error code if carry set.	Load overlay

Table 67: DOS Character Oriented Functions

Function # (AH)	Input Parameters	Output Parameters	Description
4Ch	a1- process return code		Terminate execution
4Dh		a1- return value ah- termination method.	Get child process return value

19.1.1.1 Load and Execute

The "load and execute" call requires two parameters. The first, in ds:dx, is a pointer to a zero terminated string containing the pathname of the program to execute. This must be a ".COM" or ".EXE" file and the string must contain the program name's extension. The second parameter, in es:bx, is a pointer to a LOADEXEC data structure. This data structure takes the following form:

```
LOADEXEC        struct
EnvPtr          word        ?               ;Pointer to environment area
CmdLinePtr      dword       ?               ;Pointer to command line
FCB1            dword       ?               ;Pointer to default FCB1
FCB2            dword       ?               ;Pointer to default FCB2
LOADEXEC        ends
```

Envptr is the segment address of the DOS *environment* block created for the new application. If this field contains a zero, DOS creates a copy of the current process' environment block for the child process. If the program you are running does not access the environment block, you can save several hundred bytes to a few kilobytes by pointing the environment pointer field to a string of four zeros.

The CmdLinePtr field contains the address of the command line to supply to the program. DOS will copy this command line to offset 80h in the new PSP it creates for the child process. A valid command line consists of a byte containing a character count, a least one space, any character belonging to the command line, and a terminating carriage return character (0Dh). The first byte should contain the length of the ASCII characters in the command line, not including the carriage return. If this byte contains zero, then the second byte of the command line should be the carriage return, not a space. Example:

```
MyCmdLine       byte        12, " file1 file2",cr
```

The FCB1 and FCB2 fields need to point at the two default *file control blocks* for this program. FCBs became obsolete with DOS 2.0, but Microsoft has kept FCBs around for compatibility anyway. For most programs you can point both of these fields at the following string of bytes:

```
DfltFCB         byte        3," ",0,0,0,0,0
```

The load and execute call will fail if there is insufficient memory to load the child process. When you create an ".EXE" file using MASM, it creates an executable file that grabs all available memory, by default. Therefore, there will be *no* memory available for the child process and DOS will always return an error. Therefore, you must readjust the memory allocation for the parent process before attempting to run the child process. The section "Semiresident Programs" on page 1055 describes how to do this.

There are other possible errors as well. For example, DOS might not be able to locate the program name you specify with the zero terminated string. Or, perhaps, there are too many open files and DOS doesn't have a free buffer available for the file I/O. If an error occurs, DOS returns with the carry flag set and an appropriate error code in the ax register. The following example program executes the "COMMAND.COM" program, allowing a user to execute DOS commands from inside your application. When the user types "exit" at the DOS command line, DOS returns control to your program.

```
; RUNDOS.ASM - Demonstrates how to invoke a copy of the COMMAND.COM
;              DOS command line interpreter from your programs.

                include    stdlib.a
```

```
                includelib stdlib.lib

dseg            segment    para public 'data'

; MS-DOS EXEC structure.

ExecStruct      word       0                ;Use parent's Environment blk.
                dword      CmdLine          ;For the cmd ln parms.
                dword      DfltFCB
                dword      DfltFCB

DfltFCB         byte       3," ",0,0,0,0,0
CmdLine         byte       0, 0dh           ;Cmd line for program.
PgmName         dword      filename         ;Points at pgm name.

filename        byte       "c:\command.com",0

dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

Main            proc
                mov        ax, dseg         ;Get ptr to vars segment
                mov        ds, ax

                MemInit                     ;Start the memory mgr.

; Okay, we've built the MS-DOS execute structure and the necessary
; command line, now let's see about running the program.
; The first step is to free up all the memory that this program
; isn't using. That would be everything from zzzzzzseg on.
;
; Note: unlike some previous examples in other chapters, it is okay
; to call Standard Library routines in this program after freeing
; up memory. The difference here is that the Standard Library
; routines are loaded early in memory and we haven't free up the
; storage they are sitting in.

                mov        ah, 62h          ;Get our PSP value
                int        21h
                mov        es, bx
                mov        ax, zzzzzzseg    ;Compute size of
                sub        ax, bx           ; resident run code.
                mov        bx, ax
                mov        ah, 4ah          ;Release unused memory.
                int        21h


; Tell the user what is going on:

                print
                byte       cr,lf
                byte       "RUNDOS- Executing a copy of command.com",cr,lf
                byte       "Type 'EXIT' to return control to RUN.ASM",cr,lf
                byte       0

; Warning! No Standard Library calls after this point. We've just
; released the memory that they're sitting in. So the program load
; we're about to do will wipe out the Standard Library code.

                mov        bx, seg ExecStruct
                mov        es, bx
                mov        bx, offset ExecStruct ;Ptr to program record.
                lds        dx, PgmName
                mov        ax, 4b00h        ;Exec pgm
                int        21h

; In MS-DOS 6.0 the following code isn't required. But in various older
; versions of MS-DOS, the stack is messed up at this point. Just to be
; safe, let's reset the stack pointer to a decent place in memory.
;
; Note that this code preserves the carry flag and the value in the
; AX register so we can test for a DOS error condition when we are done
```

```
; fixing the stack.

                mov         bx, sseg
                mov         ss, ax
                mov         sp, offset EndStk
                mov         bx, seg dseg
                mov         ds, bx

; Test for a DOS error:

                jnc         GoodCommand
                print
                byte        "DOS error #",0
                puti
                print
                byte        " while attempting to run COMMAND.COM",cr,lf
                byte        0
                jmp         Quit

; Print a welcome back message.

GoodCommand:    print
                byte        "Welcome back to RUNDOS. Hope you had fun.",cr,lf
                byte        "Now returning to MS-DOS' version of COMMAND.COM."
                byte        cr,lf,lf,0

; Return control to MS-DOS

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment     para stack 'stack'
                dw          128 dup (0)
sseg            ends

zzzzzzseg       segment     para public 'zzzzzzseg'
Heap            db          200h dup (?)
zzzzzzseg       ends
                end         Main
```

19.1.1.2 Load Program

The load and execute function gives the parent process very little control over the child process. Unless the child communicates with the parent process via a trap or interrupt, DOS suspends the parent process until the child terminates. In many cases the parent program may want to load the application code and then execute some additional operations before the child process takes over. Semiresident programs, appearing in the previous chapter, provide a good example. The DOS "load program" function provides this capability; it will load a program from the disk and return control back to the parent process. The parent process can do whatever it feels is appropriate before passing control to the child process.

The load program call requires parameters that are very similar to the load and execute call. Indeed, the only difference is the use of the LOAD structure rather than the LOADEXEC structure, and even these structures are very similar to one another. The LOAD data structure includes two extra fields not present in the LOADEXE structure:

```
LOAD            struct
EnvPtr          word        ?               ;Pointer to environment area.
CmdLinePtr      dword       ?               ;Pointer to command line.
FCB1            dword       ?               ;Pointer to default FCB1.
FCB2            dword       ?               ;Pointer to default FCB2.
SSSP            dword       ?               ;SS:SP value for child process.
CSIP            dword       ?               ;Initial program starting point.
LOAD            ends
```

The LOAD command is useful for many purposes. Of course, this function provides the primary vehicle for creating semiresident programs; however, it is also quite useful for providing extra error recovery,

redirecting application I/O, and loading several executable processes into memory for concurrent execution.

After you load a program using the DOS load command, you can obtain the PSP address for that program by issuing the DOS get PSP address call (see "MS-DOS, PC-BIOS, and File I/O" on page 699). This would allow the parent process to modify any values appearing in the child process' PSP prior to its execution. DOS stores the termination address for a procedure in the PSP. This termination address normally appears in the double word at offset 10h in the PSP. *If you do not change this location, the program will return to the first instruction beyond the int 21h instruction for the load function.* Therefore, before actually transferring control to the user application, you should change this termination address.

19.1.1.3 Loading Overlays

Many programs contain blocks of code that are independent of one other; that is, while routines in one block of code execute, the program will not call routines in the other independent blocks of code. For example, a modern game may contain some initialization code, a "staging area" where the user chooses certain options, an "action area" where the user plays the game, and a "debriefing area" that goes over the player's actions. When running in a 640K MS-DOS machine, all this code may not fit into available memory at the same time. To overcome this memory limitation, most large programs use *overlays*. An overlay is a portion of the program code that shares memory for its code with other code modules. The DOS load overlay function provides support for large programs that need to use overlays.

Like the load and load/execute functions, the load overlay expects a pointer to the code file's pathname in the `ds:dx` register pair and the address of a data structure in the `es:bx` register pair. This overlay data structure has the following format:

```
overlay       struct
StartSeg      word       ?
RelocFactor   word       0
overlay       ends
```

The `StartSeg` field contains the segment address where you want DOS to load the program. The `RelocFactor` field contains a relocation factor. This value should be zero unless you want the starting offset of the segment to be something other than zero.

19.1.1.4 Terminating a Process

The process termination function is nothing new to you by now, you've used this function over and over again already if you written any assembly language programs and run them under DOS (the Standard Library `ExitPgm` macro executes this command). In this section we'll look at exactly what the terminate process function call does.

First of all, the terminate process function gives you the ability to pass a single byte *termination code* back to the parent process. Whatever value you pass in al to the terminate call becomes the return, or termination code. The parent process can test this value using the Get Child Process Return Value call (see the next section). You can also test this return value in a DOS batch file using the "if errorlevel" statement.

The terminate process command does the following:

- Flushes file buffers and closes files.
- Restores the termination address (int 22h) from offset 0Ah in the PSP (this is the return address of the process).
- Restores the address of the Break handler (int 23h) from offset 0Eh in the PSP (see "Exception Handling in DOS: The Break Handler" on page 1070)
- Restores the address of the critical error handler (int 24h) from offset 12h in the PSP (see "Exception Handling in DOS: The Critical Error Handler" on page 1071).

- Deallocates any memory held by the process.

Unless you *really* know what you're doing, you should not change the values at offsets 0Ah, 0Eh, or 12h in the PSP. By doing so you could produce an inconsistent system when your program terminates.

19.1.1.5 Obtaining the Child Process Return Code

A parent process can obtain the return code from a child process by making the DOS Get Child Process Return Code function call. This call returns the value in the al register at the point of termination plus information that tells you how the child process terminated.

This call (ah=4Dh) returns the termination code in the al register. It also returns the cause of termination in the ah register. The ah register will contain one of the following values:

Table 68: Termination Cause

Value in AH	Reason for Termination
0	Normal termination (int 21h, ah=4Ch)
1	Terminated by ctrl-C
2	Terminated by critical error
3	TSR termination (int 21h, ah=31h)

The termination code appearing in al is valid only for normal and TSR terminations.

Note that you can only call this routine *once* after a child process terminates. MS-DOS returns meaningless values in AX after the first such call. Likewise, if you use this function without running a child process, the results you obtain will be meaningless. DOS does not return if you do this.

19.1.2 Exception Handling in DOS: The Break Handler

Whenever the users presses a ctrl-C or ctrl-Break key MS-DOS may trap such a key sequence and execute an `int 23h` instruction[1]. MS-DOS provides a default break handler routine that terminates the program. However, a well-written program generally replaces the default break handler with one of its own so it can capture ctrl-C or ctrl-break key sequences and shut the program down in an orderly fashion.

When DOS terminates a program due to a break interrupt, it flushes file buffers, closes all open files, releases memory belonging to the application, all the normal stuff it does on program termination. However, it does *not* restore any interrupt vectors (other than interrupt 23h and interrupt 24h). If your code has replaced any interrupt vectors, especially hardware interrupt vectors, then those vectors will still be pointing at your program's interrupt service routines after DOS terminates your program. This will probably crash the system when DOS loads a new program over the top of your code. Therefore, you should write a break handler so your application can shut itself down in an orderly fashion if the user presses ctrl-C or ctrl-break.

The easiest, and perhaps most universal, break handler consists of a single instruction – `iret`. If you point the interrupt 23h vector at an `iret` instruction, MS-DOS will simply ignore any ctrl-C or ctrl-break keys you press. This is very useful for turning off the break handling during critical sections of code that you do not want the user to interrupt.

1. MS-DOS always executes an int 23h instruction if it is processing a function code in the range 1-0Ch. For other DOS functions, MS-DOS only executes int 23h if the Break flag is set

On the other hand, simply turning off ctrl-C and ctrl-break handling throughout your entire program is not satisfactory either. If for some reason the user wants to abort your program, pressing ctrl-break or ctrl-C is what they will probably try to do this. If your program disallows this, the user may resort to something more drastic like ctrl-alt-delete to reset the machine. This will certainly mess up any open files and may cause other problems as well (of course, you don't have to worry about restoring any interrupt vectors!).

To patch in your own break handler is easy – just store the address of your break handler routine into the interrupt vector 23h. You don't even have to save the old value, DOS does this for you automatically (it stores the original vector at offset 0Eh in the PSP). Then, when the users presses a ctrl-C or ctrl-break key, MS-DOS transfers control to your break handler.

Perhaps the best response for a break handler is to set some flag to tell the application and break occurred, and then leave it up to the application to test this flag a reasonable points to determine if it should shut down. Of course, this does require that you test this flag at various points throughout your application, increasing the complexity of your code. Another alternative is to save the original int 23h vector and transfer control to DOS' break handler after you handle important operations yourself. You can also write a specialized break handler to return a DOS termination code that the parent process can read.

Of course, there is no reason you cannot change the interrupt 23h vector at various points throughout your program to handle changing requirements. At various points you can disable the break interrupt entirely, restore interrupt vectors at others, or prompt the user at still other points.

19.1.3 Exception Handling in DOS: The Critical Error Handler

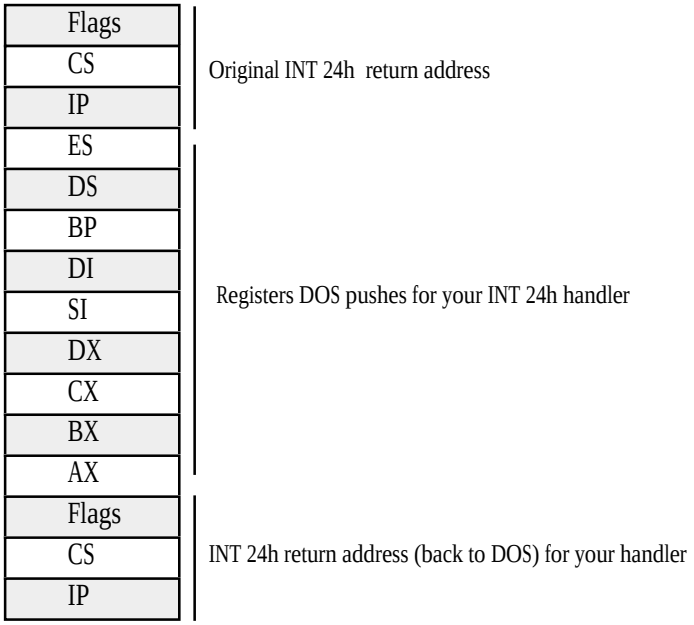
DOS invokes the critical error handler by executing an int 24h instruction whenever some sort of I/O error occurs. The default handler prints the familiar message:

```
I/O Device Specific Error Message
Abort, Retry, Ignore, Fail?
```

If the user presses an "A", this code immediately returns to DOS' COMMAND.COM program; *it doesn't even close any open files*. If the user presses an "R" to retry, MS-DOS will retry the I/O operation, though this usually results in another call to the critical error handler. The "I" option tells MS-DOS to ignore the error and return to the calling program as though nothing had happened. An "F" response instructs MS-DOS to return an error code to the calling program and let it handle the problem.

Of the above options, having the user press "A" is the most dangerous. This causes an immediate return to DOS and your code does not get the chance to clean up anything. For example, if you've patched some interrupt vectors, your program will not get the opportunity to restore them if the user selects the abort option. This may crash the system when MS-DOS loads the next program over the top of your interrupt service routine(s) in memory.

To intercept DOS critical errors, you will need to patch the interrupt 24h vector to point at your own interrupt service routine. Upon entry into your interrupt 24h service routine, the stack will contain the following data:



Stack Contents Upon Entry to a Critical Error Handler

MS-DOS passes important information in several of the registers to your critical error handler. By inspecting these values you can determine the cause of the critical error and the device on which it occurred. The high order bit of the ah register determines if the error occurred on a block structured device (typically a disk or tape) or a character device. The other bits in ah have the following meaning:

Table 69: Device Error Bits in AH

Bit(s)	Description
0	0=Read operation. 1=Write operation.
1-2	Indicates affected disk area. 00- MS-DOS area. 01- File allocation table (FAT). 10- Root directory. 11- Files area.
3	0- Fail response not allowed. 1- Fail response is okay.
4	0- Retry response not allowed. 1- Retry response is okay.
5	0- Ignore response is not allowed. 1- Ignore response is okay.
6	Undefined
7	0- Character device error. 1- Block structured device error.

In addition to the bits in ah, for block structured devices the al register contains the drive number where the error occurred (0=A, 1=B, 2=C, etc.). The value in the al register is undefined for character devices.

The lower half of the di register contains additional information about the block device error (the upper byte of di is undefined, you will need to mask out those bits before attempting to test this data).

Table 70: Block Structured Device Error Codes (in L.O. byte of DI)

Error Code	Description
0	Write protection error.
1	Unknown drive.
2	Drive not ready.
3	Invalid command.
4	Data error (CRC error).
5	Length of request structure is incorrect.
6	Seek error on device.
7	Disk is not formatted for MS-DOS.
8	Sector not found.
9	Printer out of paper.
0Ah	Write error.
0Bh	Read error.
0Ch	General failure.
0Fh	Disk was changed at inappropriate time.

Upon entry to your critical error handler, interrupts are turned off. Because this error occurs as a result of some MS-DOS call, MS-DOS is already entered and you will not be able to make any calls other than functions 1-0Ch and 59h (get extended error information).

Your critical error handler must preserve all registers except al. The handler must return to DOS with an iret instruction and al must contain one of the following codes:

Table 71: Critical Error Handler Return Codes

Code	Meaning
0	Ignore device error.
1	Retry I/O operation again.
2	Terminate process (abort).
3	Fail current system call.

The following code provides a trivial example of a critical error handler. The main program attempts to send a character to the printer. If you do not connect a printer, or turn off the printer before running this program, it will generate the critical error.

```
; Sample INT 24h critical error handler.
;
; This code demonstrates a sample critical error handler.
; It patches into INT 24h and displays an appropriate error
; message and asks the user if they want to retry, abort, ignore,
; or fail (just like DOS).
```

```
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment    para public 'data'

Value           word       0
ErrCode         word       0

dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

; A replacement critical error handler. Note that this routine
; is even worse than DOS', but it demonstrates how to write
; such a routine. Note that we cannot call any Standard Library
; I/O routines in the critical error handler because they do not
; use DOS calls 1-0Ch, which are the only allowable DOS calls at
; this point.


CritErrMsg      byte       cr,lf
                byte       "DOS Critical Error!",cr,lf
                byte       "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24         proc       far
                push       dx
                push       ds
                push       ax

                push       cs
                pop        ds
Int24Lp:        lea        dx, CritErrMsg
                mov        ah, 9             ;DOS print string call.
                int        21h

                mov        ah, 1             ;DOS read character call.
                int        21h
                and        al, 5Fh           ;Convert l.c. -> u.c.

                cmp        al, 'I'           ;Ignore?
                jne        NotIgnore
                pop        ax
                mov        al, 0
                jmp        Quit24

NotIgnore:      cmp        al, 'r'           ;Retry?
                jne        NotRetry
                pop        ax
                mov        al, 1
                jmp        Quit24

NotRetry:       cmp        al, 'A'           ;Abort?
                jne        NotAbort
                pop        ax
                mov        al, 2
                jmp        Quit24

NotAbort:       cmp        al, 'F'
                jne        BadChar
                pop        ax
                mov        al, 3
Quit24:         pop        ds
                pop        dx
                iret

BadChar:        mov        ah, 2
                mov        dl, 7             ;Bell character
                jmp        Int24Lp
MyInt24         endp
```

```
Main            proc
                mov         ax, dseg
                mov         ds, ax
                mov         es, ax
                meminit

                mov         ax, 0
                mov         es, ax
                mov         word ptr es:[24h*4], offset MyInt24
                mov         es:[24h*4 + 2], cs

                mov         ah, 5
                mov         dl, 'a'
                int         21h
                rcl         Value, 1
                and         Value, 1
                mov         ErrCode, ax
                printf
                byte        cr,lf,lf
                byte        "Print char returned with error status %d and "
                byte        "error code %d\n",0
                dword       Value, ErrCode

Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends


; Allocate a reasonable amount of space for the stack (8k).
; Note: if you use the pattern matching package you should set up a
;       somewhat larger stack.

sseg            segment     para stack 'stack'
stk             db          1024 dup ("stack ")
sseg            ends


; zzzzzzseg must be the last segment that gets loaded into memory!
; This is where the heap begins.

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

19.1.4 Exception Handling in DOS: Traps

In addition to the break and critical error exceptions, there are the 80x86 exceptions that can happen during the execution of your programs. Examples include the divide error exception, bounds exception, and illegal opcode exception. A well-written application will always handle all possible exceptions.

DOS does not provide direct support for these exceptions, other than a possible default handler. In particular, DOS does not restore such vectors when the program terminates; this is something the application, break handler, and critical error handler must take care of. For more information on these exceptions, see "Exceptions" on page 1000.

19.1.5 Redirection of I/O for Child Processes

When a child process begins execution, it inherits all open files from the parent process (with the exception of certain files opened with networking file functions). In particular, this includes the default

files opened for the DOS *standard input*, *standard output*, *standard error*, *auxiliary*, and *printer* devices. DOS assigns the file handle values zero through four, respectively, to these devices. If a parent process closes one of these file handles and then reassigns the handle with a Force Duplicate File Handle call.

Note that the DOS EXEC call does not process the I/O redirection operators ("<", and ">", and "|"). If you want to redirect the standard I/O of a child process, you must do this before loading and executing the child process. To redirect one of the five standard I/O devices, you should do the following steps:

1) Duplicate the file handle you want to redirect (e.g., to redirect the standard output, duplicate file handle one).

2) Close the affected file (e.g., file handle one for standard output).

3) Open a file using the standard DOS Create or CreateNew calls.

4) Use the Force Duplicate File Handle call to copy the new file handle to file handle one.

5) Run the child process.

6) On return from the child, close the file.

7) Copy the file handle you duplicated in step one back to the standard output file handle using the Force Duplicate Handle function.

This technique looks like it would be perfect for redirecting printer or serial port I/O. Unfortunately, many programs bypass DOS when sending data to the printer and use the BIOS call or, worse yet, go directly to the hardware. Almost no software bothers with DOS' serial port support – it truly is that bad. However, most programs *do* call DOS to input or output characters on the standard input, output, and error devices. The following code demonstrates how to redirect the output of a child process to a file.

```
; REDIRECT.ASM -Demonstrates how to redirect I/O for a child process.
; This particular program invokes COMMAND.COM to execute
; a DIR command, when is sent to the specified output file.

                include    stdlib.a
                includelib stdlib.lib

dseg            segment    para public 'data'

OrigOutHandle   word       ?                ;Holds copy of STDOUT handle.
FileHandle      word       ?                ;File I/O handle.
FileName        byte       "dirctry.txt",0  ;Filename for output data.

; MS-DOS EXEC structure.

ExecStruct      word       0                ;Use parent's Environment blk.
                dword      CmdLine          ;For the cmd ln parms.
                dword      DfltFCB
                dword      DfltFCB

DfltFCB         byte       3," ",0,0,0,0,0
CmdLine         byte       7, " /c DIR", 0dh ;Do a directory command.
PgmName         dword      PgmNameStr       ;Points at pgm name.
PgmNameStr      byte       "c:\command.com",0
dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg


Main            proc
                mov        ax, dseg         ;Get ptr to vars segment
                mov        ds, ax
                MemInit                     ;Start the memory mgr.

; Free up some memory for COMMAND.COM:

                mov        ah, 62h          ;Get our PSP value
                int        21h
```

```
                mov     es, bx
                mov     ax, zzzzzzseg   ;Compute size of
                sub     ax, bx          ; resident run code.
                mov     bx, ax
                mov     ah, 4ah         ;Release unused memory.
                int     21h

; Save original output file handle.

                mov     bx, 1           ;Std out is file handle 1.
                mov     ah, 45h         ;Duplicate the file handle.
                int     21h
                mov OrigOutHandle, ax;Save duplicate handle.

; Open the output file:

                mov     ah, 3ch         ;Create file.
                mov     cx, 0           ;Normal attributes.
                lea     dx, FileName
                int     21h
                mov     FileHandle, ax  ;Save opened file handle.

; Force the standard output to send its output to this file.
; Do this by forcing the file's handle onto file handle #1 (stdout).

                mov     ah, 46h         ;Force dup file handle
                mov     cx, 1           ;Existing handle to change.
                mov     bx, FileHandle  ;New file handle to use.
                int     21h

; Print the first line to the file:

                print
                byte    "Redirected directory listing:",cr,lf,0

; Okay, execute the DOS DIR command (that is, execute COMMAND.COM with
; the command line parameter "/c DIR").

                mov     bx, seg ExecStruct
                mov     es, bx
                mov     bx, offset ExecStruct ;Ptr to program record.
                lds     dx, PgmName
                mov     ax, 4b00h       ;Exec pgm
                int     21h

                mov     bx, sseg        ;Reset the stack on return.
                mov     ss, ax
                mov     sp, offset EndStk
                mov     bx, seg dseg
                mov     ds, bx

; Okay, close the output file and switch standard output back to the
; console.

                mov     ah, 3eh         ;Close output file.
                mov     bx, FileHandle
                int     21h

                mov     ah, 46h         ;Force duplicate handle
                mov     cx, 1           ;StdOut
                mov     bx, OrigOutHandle ;Restore previous handle.
                int     21h

; Return control to MS-DOS

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment stack 'stack'
                dw      128 dup (0)
endstk          dw      ?
sseg            ends
```

```
zzzzzzseg       segment     para public 'zzzzzzseg'
Heap            db          200h dup (?)
zzzzzzseg       ends
                end         Main
```

19.2 Shared Memory

The only problem with running different DOS programs as part of a single application is *interprocess communication*. That is, how do all these programs talk to one other? When a typical DOS application runs, DOS loads in all code and data segments; there is no provision, other than reading data from a file or the process termination code, for one process to pass information to another. Although file I/O will work, it is cumbersome and slow. The ideal solution would be for one process to leave a copy of various variables that other processes can share. Your programs can easily do this using *shared memory*.

Most modern multitasking operating systems provide for shared memory – memory that appears in the address space of two or more processes. Furthermore, such shared memory is often *persistent*, meaning it continues to hold values after its creator process terminates. This allows other processes to start later and use the values left behind by the shared variables' creator.

Unfortunately, MS-DOS is not a modern multitasking operating system and it does not support shared memory. However, we can easily write a resident program that provides this capability missing from DOS. The following sections describe how to create two types of shared memory regions – static and dynamic.

19.2.1 Static Shared Memory

A TSR to implement *static shared memory* is trivial. It is a passive TSR that provides three functions – verify presence, remove, and return segment pointer. The transient portion simply allocates a 64K data segment and then terminates. Other processes can obtain the address of the 64K shared memory block by making the "return segment pointer" call. These processes can place all their shared data into the segment belonging to the TSR. When one process quits, the shared segment remains in memory as part of the TSR. When a second process runs and links with the shared segment, the variables from the shared segment are still intact, so the new process can access those values. When all processes are done sharing data, the user can remove the shared memory TSR with the remove function.

As mentioned above, there is almost nothing to the shared memory TSR. The following code implements it:

```
; SHARDMEM.ASM
;
; This TSR sets aside a 64K shared memory region for other processes to use.
;
; Usage:
;
;       SHARDMEM -          Loads resident portion and activates
;                           shared memory capabilities.
;
;       SHARDMEM REMOVE -   Removes shared memory TSR from memory.
;
; This TSR checks to make sure there isn't a copy already active in
; memory. When removing itself from memory, it makes sure there are
; no other interrupts chained into INT 2Fh before doing the remove.
;
;
;
; The following segments must appear in this order and before the
; Standard Library includes.

ResidentSeg     segment     para public 'Resident'
ResidentSeg     ends

SharedMemory    segment     para public 'Shared'
```

```
SharedMemory  ends

EndResident   segment   para public 'EndRes'
EndResident   ends

              .xlist
              .286
              include   stdlib.a
              includelib stdlib.lib
              .list


; Resident segment that holds the TSR code:

ResidentSeg   segment   para public 'Resident'
              assume    cs:ResidentSeg, ds:nothing

; Int 2Fh ID number for this TSR:

MyTSRID       byte      0
              byte      0             ;Padding so we can print it.

; PSP is the psp address for this program.

PSP           word      0

OldInt2F      dword     ?


; MyInt2F-    Provides int 2Fh (multiplex interrupt) support for this
;             TSR. The multiplex interrupt recognizes the following
;             subfunctions (passed in AL):
;
;             00h- Verify presence.    Returns 0FFh in AL and a pointer
;                                      to an ID string in es:di if the
;                                      TSR ID (in AH) matches this
;                                      particular TSR.
;
;             01h- Remove.             Removes the TSR from memory.
;                                      Returns 0 in AL if successful,
;                                      1 in AL if failure.
;
;             10h- Return Seg Adrs.    Returns the segment address of the
;                                      shared segment in ES.

MyInt2F       proc      far
              assume    ds:nothing

              cmp       ah, MyTSRID       ;Match our TSR identifier?
              je        YepItsOurs
              jmp       OldInt2F

; Okay, we know this is our ID, now check for a verify, remove, or
; return segment call.

YepItsOurs:   cmp       al, 0             ;Verify Call
              jne       TryRmv
              mov       al, 0ffh          ;Return success.
              lesi      IDString
              iret                        ;Return back to caller.

IDString      byte      "Static Shared Memory TSR",0

TryRmv:       cmp       al, 1             ;Remove call.
              jne       TryRetSeg

; See if we can remove this TSR:

              push      es
              mov       ax, 0
              mov       es, ax
              cmp       word ptr es:[2Fh*4], offset MyInt2F
              jne       TRDone
              cmp       word ptr es:[2Fh*4 + 2], seg MyInt2F
```

```
                jе      CanRemove;Branch if we can.
TRDone:         mov     ax, 1           ;Return failure for now.
                pop     es
                iret

; Okay, they want to remove this guy *and* we can remove it from memory.
; Take care of all that here.

                assume  ds:ResidentSeg

CanRemove:      push    ds
                pusha
                cli                     ;Turn off the interrupts while
                mov     ax, 0           ; we mess with the interrupt
                mov     es, ax          ; vectors.
                mov     ax, cs
                mov     ds, ax

                mov     ax, word ptr OldInt2F
                mov     es:[2Fh*4], ax
                mov     ax, word ptr OldInt2F+2
                mov     es:[2Fh*4 + 2], ax


; Okay, one last thing before we quit- Let's give the memory allocated
; to this TSR back to DOS.

                mov     ds, PSP
                mov     es, ds:[2Ch]    ;Ptr to environment block.
                mov     ah, 49h         ;DOS release memory call.
                int     21h

                mov     ax, ds          ;Release program code space.
                mov     es, ax
                mov     ah, 49h
                int     21h

                popa
                pop     ds
                pop     es
                mov     ax, 0           ;Return Success.
                iret

; See if they want us to return the segment address of our shared segment
; here.

TryRetSeg:      cmp     al, 10h         ;Return Segment Opcode
                jne IllegalOp
                mov     ax, SharedMemory
                mov     es, ax
                mov     ax, 0           ;Return success
                clc
                iret

; They called us with an illegal subfunction value. Try to do as little
; damage as possible.

IllegalOp:      mov     ax, 0           ;Who knows what they were thinking?
                iret
MyInt2F         endp
                assume  ds:nothing
ResidentSeg     ends


; Here's the segment that will actually hold the shared data.

SharedMemory    segment para public 'Shared'
                db      0FFFFh dup (?)
SharedMemory    ends


cseg            segment para public 'code'
                assume  cs:cseg, ds:ResidentSeg
```

```
; SeeIfPresent-         Checks to see if our TSR is already present in memory.
;                       Sets the zero flag if it is, clears the zero flag if
;                       it is not.

SeeIfPresent  proc      near
              push      es
              push      ds
              push      di
              mov       cx, 0ffh          ;Start with ID 0FFh.
IDLoop:       mov       ah, cl
              push      cx
              mov       al, 0             ;Verify presence call.
              int       2Fh
              pop       cx
              cmp       al, 0             ;Present in memory?
              je        TryNext
              strcmpl
              byte      "Static Shared Memory TSR",0
              je        Success

TryNext:      dec       cl                ;Test USER IDs of 80h..FFh
              js        IDLoop
              cmp       cx, 0             ;Clear zero flag.
Success:      pop       di
              pop       ds
              pop       es
              ret
SeeIfPresent  endp



; FindID-     Determines the first (well, last actually) TSR ID available
;             in the multiplex interrupt chain. Returns this value in
;             the CL register.
;
;             Returns the zero flag set if it locates an empty slot.
;             Returns the zero flag clear if failure.

FindID        proc      near
              push      es
              push      ds
              push      di

              mov       cx, 0ffh          ;Start with ID 0FFh.
IDLoop:       mov       ah, cl
              push      cx
              mov       al, 0             ;Verify presence call.
              int       2Fh
              pop       cx
              cmp       al, 0             ;Present in memory?
              je        Success
              dec       cl                ;Test USER IDs of 80h..FFh
              js        IDLoop
              xor       cx, cx
              cmp       cx, 1             ;Clear zero flag
Success:      pop       di
              pop       ds
              pop       es
              ret
FindID        endp



Main          proc
              meminit

              mov       ax, ResidentSeg
              mov       ds, ax

              mov       ah, 62h           ;Get this program's PSP
              int       21h               ; value.
              mov       PSP, bx

; Before we do anything else, we need to check the command line
```

```
; parameters. If there is one, and it is the word "REMOVE", then remove
; the resident copy from memory using the multiplex (2Fh) interrupt.

                argc
                cmp         cx, 1               ;Must have 0 or 1 parms.
                jb          TstPresent
                je          DoRemove
Usage:          print
                byte        "Usage:",cr,lf
                byte        " shardmem",cr,lf
                byte        "or shardmem REMOVE",cr,lf,0
                ExitPgm


; Check for the REMOVE command.

DoRemove:       mov         ax, 1
                argv
                stricmpl
                byte        "REMOVE",0
                jne         Usage

                call        SeeIfPresent
                je          RemoveIt
                print
                byte        "TSR is not present in memory, cannot remove"
                byte        cr,lf,0
                ExitPgm

RemoveIt:       mov         MyTSRID, cl
                printf
                byte        "Removing TSR (ID #%d) from memory...",0
                dword       MyTSRID

                mov         ah, cl
                mov         al, 1               ;Remove cmd, ah contains ID
                int         2Fh
                cmp         al, 1               ;Succeed?
                je          RmvFailure
                print
                byte        "removed.",cr,lf,0
                ExitPgm

RmvFailure:     print
                byte        cr,lf
                byte        "Could not remove TSR from memory.",cr,lf
                byte        "Try removing other TSRs in the reverse order "
                byte        "you installed them.",cr,lf,0
                ExitPgm



; Okay, see if the TSR is already in memory. If so, abort the
; installation process.

TstPresent:     call        SeeIfPresent
                jne         GetTSRID
                print
                byte        "TSR is already present in memory.",cr,lf
                byte        "Aborting installation process",cr,lf,0
                ExitPgm


; Get an ID for our TSR and save it away.

GetTSRID:       call        FindID
                je          GetFileName
                print
                byte        "Too many resident TSRs, cannot install",cr,lf,0
                ExitPgm


; Things look cool so far, so install the interrupts
```

```
GetFileName:    mov         MyTSRID, cl
                print
                byte        “Installing interrupts...”,0


; Patch into the INT 2Fh interrupt chain.

                cli                             ;Turn off interrupts!
                mov         ax, 0
                mov         es, ax
                mov         ax, es:[2Fh*4]
                mov         word ptr OldInt2F, ax
                mov         ax, es:[2Fh*4 + 2]
                mov         word ptr OldInt2F+2, ax
                mov         es:[2Fh*4], offset MyInt2F
                mov         es:[2Fh*4+2], seg ResidentSeg
                sti                             ;Okay, ints back on.

; We’re hooked up, the only thing that remains is to zero out the shared
; memory segment and then terminate and stay resident.

                printf
                byte        “Installed, TSR ID #%d.”,cr,lf,0
                dword       MyTSRID

                mov         ax, SharedMemory    ;Zero out the shared
                mov         es, ax              ; memory segment.
                mov         cx, 32768           ;32K words = 64K bytes.
                xor         ax, ax              ;Store all zeros,
                mov         di, ax              ; starting at offset zero.
        rep     stosw


                mov         dx, EndResident     ;Compute size of program.
                sub         dx, PSP
                mov         ax, 3100h           ;DOS TSR command.
                int         21h
Main            endp
cseg            ends

sseg            segment     para stack ‘stack’
stk             db          256 dup (?)
sseg            ends

zzzzzzseg       segment     para public ‘zzzzzz’
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

This program simply carves out a chunk of memory (the 64K in the SharedMemory segment) and returns a pointer to it in es whenever a program executes the appropriate int 2Fh call (ah= TSR ID and al=10h). The only catch is how do we declared shared variables in the applications that use shared memory? Well, that’s fairly easy if we play a sneaky trick on MASM, the Linker, DOS, and the 80x86.

When DOS loads your program into memory, it generally loads the segments in the same order they first appear in your source files. The UCR Standard Library, for example, takes advantage of this by insisting that you include a segment named zzzzzzseg at the end of all your assembly language source files. The UCR Standard Library memory management routines build the heap starting at zzzzzzseg, it must be the last segment (containing valid data) because the memory management routines may overwrite anything following zzzzzzseg.

For our shared memory segment, we would like to create a segment something like the following:

```
SharedMemory    segment     para public ‘Shared’

« define all shared variables here»

SharedMemory    ends
```

Applications that share data would define all shared variables in this shared segment. There are, however, five problems. First, how do we tell the assembler/linker/DOS/80x86 that this is a *shared* segment, rather than having a separate segment for each program? Well, this problem is easy to solve; we don't bother telling MASM, the linker, or DOS anything. The way we make the different applications all share the same segment in memory is to invoke the shared memory TSR in the code above with function code 10h. This returns the address of the TSR's SharedMemory segment in the es register. In our assembly language programs we fool MASM into thinking es points at its local shared memory segment when, in fact, es points at the global segment.

The second problem is minor, but annoying nonetheless. When you create a segment, MASM, the linker, and DOS set aside storage for that segment. If you declare a large number of variables in a shared segment, this can waste memory since the program will actually use the memory space in the global shared segment. One easy way to reclaim the storage that MASM reserves for this segment is to define the shared segment *after* zzzzzzseg in your shared memory applications. By doing so, the Standard Library will absorb any memory reserved for the (dummy) shared memory segment into the heap, since all memory after zzzzzzseg belongs to the heap (when you use the standard meminit call).

The third problem is slightly more difficult to deal with. Since you will not be use the local segment, you cannot initialize any variables in the shared memory segment by placing values in the operand field of byte, word, dword, etc., directives. Doing so will only initialize the local memory in the heap, the system will not copy this data to the global shared segment. Generally, this isn't a problem because processes won't normally initialize shared memory as they load. Instead, there will probably be a single application you run first that initializes the shared memory area for the rest of the processes that using the global shared segment.

The fourth problem is that you cannot initialize any variables with the address of an object in shared memory. For example, if the variable shared_K is in the shared memory segment, you could not use a statement like the following:

```
printf
byte       "Value of shared_K is %d\n",0
dword      shared_K
```

The problem with this code is that MASM initializes the double word after the string above with the address of the shared_K variable *in the local copy of the shared data segment*. This will not print out the copy in the global shared data segment.

The last problem is anything but minor. All programs that use the global shared memory segment *must* define their variables at identical offsets within the shared segment. Given the way MASM assigns offsets to variables within a segment, if you are one byte off in the declaration of *any* of your variables, your program will be accessing its variables at different addresses than other processes sharing the global shared segment. This will scramble memory and produce a disaster. The only reasonable way to declare variables for shared memory programs is to create an include file with all the shared variable declarations for all concerned programs. Then include this single file into all the programs that share the variables. Now you can add, remove, or modify variables without having to worry about maintaining the shared variable declarations in the other files.

The following two sample programs demonstrate the use of shared memory. The first application reads a string from the user and stuffs it into shared memory. The second application reads that string from shared memory and displays it on the screen.

First, here is the include file containing the single shared variable declaration used by both applications:

```
; shmvars.asm
;
; This file contains the shared memory variable declarations used by
; all applications that refer to shared memory.

InputLine      byte       128 dup (?)
```

Here is the first application that reads an input string from the user and shoves it into shared memory:

```
; SHMAPP1.ASM
;
; This is a shared memory application that uses the static shared memory
; TSR (SHARDMEM.ASM). This program inputs a string from the user and
; passes that string to SHMAPP2.ASM through the shared memory area.
;
;
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'
ShmID           byte       0
dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent-Checks to see if the shared memory TSR is present in memory.
;               Sets the zero flag if it is, clears the zero flag if
;               it is not. This routine also returns the TSR ID in CL.

SeeIfPresent    proc       near
                push       es
                push       ds
                push       di
                mov        cx, 0ffh          ;Start with ID 0FFh.
IDLoop:         mov        ah, cl
                push       cx
                mov        al, 0             ;Verify presence call.
                int        2Fh
                pop        cx
                cmp        al, 0             ;Present in memory?
                je         TryNext
                strcmpl
                byte       "Static Shared Memory TSR",0
                je         Success

TryNext:        dec        cl                ;Test USER IDs of 80h..FFh
                js         IDLoop
                cmp        cx, 0             ;Clear zero flag.
Success:        pop        di
                pop        ds
                pop        es
                ret
SeeIfPresent    endp



; The main program for application #1 links with the shared memory
; TSR and then reads a string from the user (storing the string into
; shared memory) and then terminates.

Main            proc
                assume     cs:cseg, ds:dseg, es:SharedMemory
                mov        ax, dseg
                mov        ds, ax
                meminit

                print
                byte       "Shared memory application #1",cr,lf,0

; See if the shared memory TSR is around:

                call       SeeIfPresent
                je         ItsThere
                print
                byte       "Shared Memory TSR (SHARDMEM) is not loaded.",cr,lf
                byte       "This program cannot continue execution.",cr,lf,0
```

```
                ExitPgm

; If the shared memory TSR is present, get the address of the shared segment
; into the ES register:

ItsThere:       mov         ah, cl              ;ID of our TSR.
                mov         al, 10h             ;Get shared segment address.
                int         2Fh

; Get the input line from the user:

                print
                byte        "Enter a string: ",0

                lea         di, InputLine       ;ES already points at proper seg.
                gets

                print
                byte        "Entered '",0
                puts
                print
                byte        "' into shared memory.",cr,lf,0


Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg ends

sseg            segment     para stack 'stack'
stk             db          1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends


; The shared memory segment must appear after "zzzzzzseg".
; Note that this isn't the physical storage for the data in the
; shared segment. It's really just a place holder so we can declare
; variables and generate their offsets appropriately. The UCR Standard
; Library will reuse the memory associated with this segment for the
; heap. To access data in the shared segment, this application calls
; the shared memory TSR to obtain the true segment address of the
; shared memory segment. It can then access variables in the shared
; memory segment (where ever it happens to be) off the ES register.
;
; Note that all the variable declarations go into an include file.
; All applications that refer to the shared memory segment include
; this file in the SharedMemory segment. This ensures that all
; shared segments have the exact same variable layout.

SharedMemory    segment     para public 'Shared'

                include     shmvars.asm

SharedMemory    ends
                end         Main
```

The second application is very similar, here it is

```
; SHMAPP2.ASM
;
; This is a shared memory application that uses the static shared memory
; TSR (SHARDMEM.ASM). This program assumes the user has already run the
; SHMAPP1 program to insert a string into shared memory. This program
; simply prints that string from shared memory.
;
```

```
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'
ShmID           byte       0
dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent Checks to see if the shared memory TSR is present in memory.
;              Sets the zero flag if it is, clears the zero flag if
;              it is not. This routine also returns the TSR ID in CL.

SeeIfPresent    proc       near
                push       es
                push       ds
                push       di
                mov        cx, 0ffh          ;Start with ID 0FFh.
IDLoop:         mov        ah, cl
                push       cx
                mov        al, 0             ;Verify presence call.
                int        2Fh
                pop        cx
                cmp        al, 0             ;Present in memory?
                je         TryNext
                strcmpl
                byte       "Static Shared Memory TSR",0
                je         Success

TryNext:        dec        cl                ;Test USER IDs of 80h..FFh
                js         IDLoop
                cmp        cx, 0             ;Clear zero flag.
Success:        pop        di
                pop        ds
                pop        es
                ret
SeeIfPresent    endp



; The main program for application #1 links with the shared memory
; TSR and then reads a string from the user (storing the string into
; shared memory) and then terminates.

Main            proc
                assume     cs:cseg, ds:dseg, es:SharedMemory
                mov        ax, dseg
                mov        ds, ax
                meminit

                print
                byte       "Shared memory application #2",cr,lf,0

; See if the shared memory TSR is around:

                call       SeeIfPresent
                je         ItsThere
                print
                byte       "Shared Memory TSR (SHARDMEM) is not loaded.",cr,lf
                byte       "This program cannot continue execution.",cr,lf,0
                ExitPgm

; If the shared memory TSR is present, get the address of the shared segment
; into the ES register:

ItsThere:       mov        ah, cl            ;ID of our TSR.
                mov        al, 10h           ;Get shared segment address.
                int        2Fh

; Print the string input in SHMAPP1:
```

```
                print
                byte        “String from SHMAPP1 is ‘”,0

                lea         di, InputLine     ;ES already points at proper seg.
                puts

                print
                byte        “‘ from shared memory.”,cr,lf,0


Quit:           ExitPgm                       ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment     para stack ‘stack’
stk             db          1024 dup (“stack “)
sseg            ends

zzzzzzseg       segment     para public ‘zzzzzz’
LastBytes       db          16 dup (?)
zzzzzzseg       ends


; The shared memory segment must appear after “zzzzzzseg”.
; Note that this isn’t the physical storage for the data in the
; shared segment. It’s really just a place holder so we can declare
; variables and generate their offsets appropriately. The UCR Standard
; Library will reuse the memory associated with this segment for the
; heap. To access data in the shared segment, this application calls
; the shared memory TSR to obtain the true segment address of the
; shared memory segment. It can then access variables in the shared
; memory segment (where ever it happens to be) off the ES register.
;
; Note that all the variable declarations go into an include file.
; All applications that refer to the shared memory segment include
; this file in the SharedMemory segment. This ensures that all
; shared segments have the exact same variable layout.

SharedMemory    segment     para public ‘Shared’

                include     shmvars.asm

SharedMemory    ends
                end         Main
```

19.2.2 Dynamic Shared Memory

Although the static shared memory the previous section describes is very useful, it does suffer from a few limitations. First of all, any program that uses the global shared segment must be aware of the location of every other program that uses the shared segment. This effectively means that the use of the shared segment is limited to a single set of cooperating processes at any one given time. You cannot have two independent sets of programs using the shared memory at the same time. Another limitation with the static system is that you must know the size of all variables when you write your program, you cannot create dynamic data structures whose size varies at run time. It would be nice, for example, to have calls like shmalloc and shmfree that let you dynamically allocate and free memory in a shared region. Fortunately, it is very easy to overcome these limitations by creating a *dynamic shared memory manager*.

A reasonable shared memory manager will have four functions: initialize, shmalloc, shmattach, and shmfree. The initialization call reclaims all shared memory in use. The shmalloc call lets a process allocate a new block of shared memory. Only one process in a group of cooperating processes makes this call. Once shmalloc allocates a block of memory, the other processes use the shmattach call to obtain the address of the shared memory block. The following code implements a dynamic shared memory manager. The code is similar to that appearing in the Standard Library except this code allows a maximum of 64K storage on the heap.

```
; SHMALLOC.ASM
;
; This TSR sets up a dynamic shared memory system.
;
; This TSR checks to make sure there isn't a copy already active in
; memory. When removing itself from memory, it makes sure there are
; no other interrupts chained into INT 2Fh before doing the remove.
;
;
;
; The following segments must appear in this order and before the
; Standard Library includes.

ResidentSeg   segment   para public 'Resident'
ResidentSeg   ends

SharedMemory  segment   para public 'Shared'
SharedMemory  ends

EndResident   segment   para public 'EndRes'
EndResident   ends

              .xlist
              .286
              include   stdlib.a
              includelib stdlib.lib
              .list


; Resident segment that holds the TSR code:

ResidentSeg   segment   para public 'Resident'
              assume    cs:ResidentSeg, ds:nothing


NULL          equ       0


; Data structure for an allocated data region.
;
; Key-   user supplied ID to associate this region with a particular set
;        of processes.
;
; Next-  Points at the next allocated block.
; Prev-  Points at the previous allocated block.
; Size-  Size (in bytes) of allocated block, not including header structure.

Region        struct
key           word      ?
next          word      ?
prev          word      ?
blksize       word      ?
Region        ends

Startmem      equ       Region ptr [0]

AllocatedList word      0             ;Points at chain of alloc'd blocks.
FreeList      word      0             ;Points at chain of free blocks.

; Int 2Fh ID number for this TSR:

MyTSRID       byte      0
              byte      0             ;Padding so we can print it.

; PSP is the psp address for this program.

PSP           word      0

OldInt2F      dword     ?


; MyInt2F-    Provides int 2Fh (multiplex interrupt) support for this
;             TSR. The multiplex interrupt recognizes the following
;             subfunctions (passed in AL):
```

```
;
;               00h- Verify presence.     Returns 0FFh in AL and a pointer
;                                         to an ID string in es:di if the
;                                         TSR ID (in AH) matches this
;                                         particular TSR.
;
;               01h- Remove.              Removes the TSR from memory.
;                                         Returns 0 in AL if successful,
;                                         1 in AL if failure.
;
;               11h- shmalloc             CX contains the size of the block
;                                          to allocate.
;                                         DX contains the key for this block.
;                                         Returns a pointer to block in ES:DI
;                                          and size of allocated block in CX.
;                                         Returns an error code in AX. Zero
;                                          is no error, one is "key already
;                                          exists," two is "insufficient
;                                          memory for request."
;
;               12h- shmfree              DX contains the key for this block.
;                                         This call frees the specified block
;                                          from memory.
;
;               13h- shminit              Initializes the shared memory system
;                                          freeing all blocks currently in
;                                          use.
;
;               14h- shmattach            DX contains the key for a block.
;                                          Search for that block and return
;                                          its address in ES:DI. AX contains
;                                          zero if successful, three if it
;                                          cannot locate a block with the
;                                          specified key.

MyInt2F         proc        far
                assume      ds:nothing

                cmp         ah, MyTSRID;Match our TSR identifier?
                je          YepItsOurs
                jmp         OldInt2F

; Okay, we know this is our ID, now check for a verify, remove, or
; return segment call.

YepItsOurs:     cmp         al, 0             ;Verify Call
                jne         TryRmv
                mov         al, 0ffh;Return success.
                lesi        IDString
                iret                          ;Return back to caller.

IDString byte   "Dynamic Shared Memory TSR",0

TryRmv:         cmp         al, 1             ;Remove call.
                jne         Tryshmalloc

; See if we can remove this TSR:

                push        es
                mov         ax, 0
                mov         es, ax
                cmp         word ptr es:[2Fh*4], offset MyInt2F
                jne         TRDone
                cmp         word ptr es:[2Fh*4 + 2], seg MyInt2F
                je          CanRemove         ;Branch if we can.
TRDone:         mov         ax, 1             ;Return failure for now.
                pop         es
                iret

; Okay, they want to remove this guy *and* we can remove it from memory.
; Take care of all that here.

                assume      ds:ResidentSeg
```

```
CanRemove:      push    ds
                pusha
                cli                     ;Turn off the interrupts while
                mov     ax, 0           ; we mess with the interrupt
                mov     es, ax          ; vectors.
                mov     ax, cs
                mov     ds, ax

                mov     ax, word ptr OldInt2F
                mov     es:[2Fh*4], ax
                mov     ax, word ptr OldInt2F+2
                mov     es:[2Fh*4 + 2], ax


; Okay, one last thing before we quit- Let's give the memory allocated
; to this TSR back to DOS.

                mov     ds, PSP
                mov     es, ds:[2Ch]    ;Ptr to environment block.
                mov     ah, 49h         ;DOS release memory call.
                int     21h

                mov     ax, ds          ;Release program code space.
                mov     es, ax
                mov     ah, 49h
                int     21h

                popa
                pop     ds
                pop     es
                mov     ax, 0           ;Return Success.
                iret



; Stick BadKey here so that it is close to its associated branch (from below).
;
; If come here, we've discovered an allocated block with the
; specified key. Return an error code (AX=1) and the size of that
; allocated block (in CX).

BadKey:         mov     cx, [bx].Region.BlkSize
                mov     ax, 1           ;Already allocated error.
                pop     bx
                pop     ds
                iret


; See if this is a shmalloc call.
; If so, on entry -
; DX contains the key.
; CX contains the number of bytes to allocate.
;
; On exit:
;
; ES:DI points at the allocated block (if successful).
; CX contains the actual size of the allocated block (>=CX on entry).
; AX contains error code, 0 if no error.

Tryshmalloc:    cmp     al, 11h         ;shmalloc function code.
                jne Tryshmfree

; First, search through the allocated list to see if a block with the
; current key number already exists. DX contains the requested key.

                assume  ds:SharedMemory
                assume  bx:ptr Region
                assume  di:ptr Region

                push    ds
                push    bx
                mov     bx, SharedMemory
                mov     ds, bx
```

```
                mov     bx, ResidentSeg:AllocatedList
                test    bx, bx          ;Anything on this list?
                je      SrchFreeList

SearchLoop:     cmp     dx, [bx].Key    ;Key exist already?
                je      BadKey
                mov     bx, [bx].Next   ;Get next region.
                test    bx, bx          ;NULL?, if not, try another
                jne     SearchLoop      ; entry in the list.

; If an allocated block with the specified key does not already exist,
; then try to allocate one from the free memory list.

SrchFreeList:   mov     bx, ResidentSeg:FreeList
                test    bx, bx          ;Empty free list?
                je      OutaMemory

FirstFitLp:     cmp     cx, [bx].BlkSize  ;Is this block big enough?
                jbe     GotBlock
                mov     bx, [bx].Next   ;If not, on to the next one.
                test    bx, bx          ;Anything on this list?
                jne     FirstFitLp

; If we drop down here, we were unable to find a block that was large
; enough to satisfy the request. Return an appropriate error

OutaMemory:     mov     cx, 0           ;Nothing available.
                mov     ax, 2           ;Insufficient memory error.
                pop     bx
                pop     ds
                iret

; If we find a large enough block, we've got to carve the new block
; out of it and return the rest of the storage to the free list. If the
; free block is at least 32 bytes larger than the requested size, we will
; do this. If the free block is less than 32 bytes larger, we will simply
; give this free block to the requesting process. The reason for the
; 32 bytes is simple: We need eight bytes for the new block's header
; (the free block already has one) and it doesn't make sense to fragment
; blocks to sizes below 24 bytes. That would only increase processing time
; when processes free up blocks by requiring more work coalescing blocks.

GotBlock:       mov     ax, [bx].BlkSize  ;Compute difference in size.
                sub     ax, cx
                cmp     ax, 32          ;At least 32 bytes left?
                jbe     GrabWholeBlk    ;If not, take this block.

; Okay, the free block is larger than the requested size by more than 32
; bytes. Carve the new block from the end of the free block (that way
; we do not have to change the free block's pointers, only the size.

                mov     di, bx
                add     di, [bx].BlkSize  ;Scoot to end, minus 8
                sub     di, cx          ;Point at new block.

                sub     [bx].BlkSize, cx  ;Remove alloc'd block and
                sub     [bx].BlkSize, 8   ; room for header.

                mov     [di].BlkSize, cx  ;Save size of block.
                mov     [di].Key, dx    ;Save key.

; Link the new block into the list of allocated blocks.

                mov     bx, ResidentSeg:AllocatedList
                mov     [di].Next, bx
                mov     [di].Prev, NULL   ;NULL previous pointer.
                test    bx, bx          ;See if it was an empty list.
                je      NoPrev
                mov     [bx].Prev, di   ;Set prev ptr for old guy.

NoPrev:         mov     ResidentSeg:AllocatedList, di
RmvDone:        add     di, 8           ;Point at actual data area.
                mov     ax, ds          ;Return ptr in es:di.
                mov     es, ax
```

```
                mov     ax, 0           ;Return success.
                pop     bx
                pop     ds
                iret


; If the current free block is larger than the request, but not by more
; that 32 bytes, just give the whole block to the user.

GrabWholeBlk:   mov     di, bx
                mov     cx, [bx].BlkSize  ;Return actual size.
                cmp     [bx].Prev, NULL   ;First guy in list?
                je      Rmv1st
                cmp     [bx].Next, NULL   ;Last guy in list?
                je      RmvLast

; Okay, this record is sandwiched between two other in the free list.
; Cut it out from among the two.

                mov     ax, [bx].Next     ;Save the ptr to the next
                mov     bx, [bx].Prev     ; item in the prev item's
                mov     [bx].Next, ax     ; next field.

                mov     ax, bx            ;Save the ptr to the prev
                mov     bx, [di].Next     ; item in the next item's
                mov     [bx].Prev, bx     ; prev field.
                jmp     RmvDone



; The block we want to remove is at the beginning of the free list.
; It could also be the only item on the free list!

Rmv1st:         mov     ax, [bx].Next
                mov     FreeList, ax      ;Remove from free list.
                jmp     RmvDone

; If the block we want to remove is at the end of the list, handle that
; down here.

RmvLast:        mov     bx, [bx].Prev
                mov     [bx].Next, NULL
                jmp     RmvDone

                assume  ds:nothing, bx:nothing, di:nothing



; This code handles the SHMFREE function.
; On entry, DX contains the key for the block to free. We need to
; search through the allocated block list and find the block with that
; key. If we do not find such a block, this code returns without doing
; anything. If we find the block, we need to add its memory to the
; free pool. However, we cannot simply insert this block on the front
; of the free list (as we did for the allocated blocks). It might
; turn out that this block we're freeing is adjacent to one or two
; other free blocks. This code has to coalesce such blocks into
; a single free block.

Tryshmfree:     cmp     al, 12h
                jne     Tryshminit


; First, search the allocated block list to see if we can find the
; block to remove. If we don't find it in the list anywhere, just return.

                assume  ds:SharedMemory
                assume  bx:ptr Region
                assume  di:ptr Region

                push    ds
                push    di
                push    bx
```

```
                mov     bx, SharedMemory
                mov     ds, bx
                mov     bx, ResidentSeg:AllocatedList

                test    bx, bx          ;Empty allocated list?
                je      FreeDone
SrchList:       cmp     dx, [bx].Key    ;Search for key in DX.
                je      FoundIt
                mov     bx, [bx].Next
                test    bx, bx          ;At end of list?
                jne     SrchList
FreeDone:       pop     bx
                pop     di              ;Nothing allocated, just
                pop     ds              ; return to caller.
                iret


; Okay, we found the block the user wants to delete. Remove it from
; the allocated list. There are three cases to consider:
; (1) it is at the front of the allocated list, (2) it is at the end of
; the allocated list, and (3) it is in the middle of the allocated list.

FoundIt:        cmp     [bx].Prev, NULL ;1st item in list?
                je      Free1st
                cmp     [bx].Next, NULL ;Last item in list?
                je      FreeLast

; Okay, we're removing an allocated item from the middle of the allocated
; list.

                mov     di, [bx].Next   ;[next].prev := [cur].prev
                mov     ax, [bx].Prev
                mov     [di].Prev, ax
                xchg    ax, di
                mov     [di].Next, ax   ;[prev].next := [cur].next
                jmp     AddFree

; Handle the case where we are removing the first item from the allocation
; list. It is possible that this is the only item on the list (i.e., it
; is the first and last item), but this code handles that case without any
; problems.

Free1st:        mov     ax, [bx].Next
                mov     ResidentSeg:AllocatedList, ax
                jmp     AddFree

; If we're removing the last guy in the chain, simply set the next field
; of the previous node in the list to NULL.

FreeLast:       mov     di, [bx].Prev
                mov     [di].Next, NULL

; Okay, now we've got to put the freed block onto the free block list.
; The free block list is sorted according to address. We have to search
; for the first free block whose address is greater than the block we've
; just freed and insert the new free block before that one. If the two
; blocks are adjacent, then we've got to merge them into a single free
; block. Also, if the block before is adjacent, we must merge it as
; well. This will coalesce all free blocks on the free list so there
; are as few free blocks as possible and those blocks are as large as
; possible.

AddFree:        mov     ax, ResidentSeg:FreeList
                test    ax, ax          ;Empty list?
                jne     SrchPosn

; If the list is empty, stick this guy on as the only entry.

                mov     ResidentSeg:FreeList, bx
                mov     [bx].Next, NULL
                mov     [bx].Prev, NULL
                jmp     FreeDone
```

```
; If the free list is not empty, search for the position of this block
; in the free list:

SrchPosn:     mov       di, ax
              cmp       bx, di
              jb        FoundPosn
              mov       ax, [di].Next
              test      ax, ax          ;At end of list?
              jne       SrchPosn

; If we fall down here, the free block belongs at the end of the list.
; See if we need to merge the new block with the old one.

              mov       ax, di
              add       ax, [di].BlkSize  ;Compute address of 1st byte
              add       ax, 8             ; after this block.
              cmp       ax, bx
              je        MergeLast

; Okay, just add the free block to the end of the list.

              mov       [di].Next, bx
              mov       [bx].Prev, di
              mov       [bx].Next, NULL
              jmp       FreeDone

; Merge the freed block with the block DI points at.

MergeLast:    mov       ax, [di].BlkSize
              add       ax, [bx].BlkSize
              add       ax, 8
              mov       [di].BlkSize, ax
              jmp       FreeDone

; If we found a free block before which we are supposed to insert
; the current free block, drop down here and handle it.

FoundPosn:    mov       ax, bx          ;Compute the address of the
              add       ax, [bx].BlkSize  ; next block in memory.
              add       ax, 8
              cmp       ax, di          ;Equal to this block?
              jne       DontMerge

; The next free block is adjacent to the one we're freeing, so just
; merge the two.

              mov       ax, [di].BlkSize  ;Merge the sizes together.
              add       ax, 8
              add       [bx].BlkSize, ax
              mov       ax, [di].Next     ;Tweak the links.
              mov       [bx].Next, ax
              mov       ax, [di].Prev
              mov       [bx].Prev, ax
              jmp       TryMergeB4

; If the blocks are not adjacent, just link them together here.

DontMerge:    mov       ax, [di].Prev
              mov       [di].Prev, bx
              mov       [bx].Prev, ax
              mov       [bx].Next, di

; Now, see if we can merge the current free block with the previous free blk.

TryMergeB4:   mov       di, [bx].Prev
              mov       ax, di
              add       ax, [di].BlkSize
              add       ax, 8
              cmp       ax, bx
              je        CanMerge
              pop       bx
              pop       di              ;Nothing allocated, just
              pop       ds              ; return to caller.
              iret
```

```
; If we can merge the previous and current free blocks, do that here:

CanMerge:     mov       ax, [bx].Next
              mov       [di].Next, ax
              mov       ax, [bx].BlkSize
              add       ax, 8
              add       [di].BlkSize, ax
              pop       bx
              pop       di
              pop       ds
              iret

              assume    ds:nothing
              assume    bx:nothing
              assume    di:nothing

; Here's where we handle the shared memory initializatin (SHMINIT) function.
; All we got to do is create a single block on the free list (which is all
; available memory), empty out the allocated list, and then zero out all
; shared memory.

Tryshminit:   cmp       al, 13h
              jne       TryShmAttach

; Reset the memory allocation area to contain a single, free, block of
; memory whose size is 0FFF8h (need to reserve eight bytes for the block's
; data structure).

              push      es
              push      di
              push      cx

              mov       ax, SharedMemory  ;Zero out the shared
              mov       es, ax            ; memory segment.
              mov       cx, 32768
              xor       ax, ax
              mov       di, ax
        rep   stosw


; Note: the commented out lines below are unnecessary since the code above
; has already zeroed out the entire shared memory segment.
; Note: we cannot put the first record at offset zero because offset zero
; is the special value for the NULL pointer. We'll use 4 instead.

              mov       di, 4
;             mov       es:[di].Region.Key, 0 ;Key is arbitrary.
;             mov       es:[di].Region.Next, 0 ;No other entries.
;             mov       es:[di].Region.Prev, 0 ; Ditto.
              mov       es:[di].Region.BlkSize, 0FFF8h ;Rest of segment.
              mov       ResidentSeg:FreeList, di

              pop       cx
              pop       di
              pop       es
              mov       ax, 0             ;Return no error.
              iret


; Handle the SHMATTACH function here. On entry, DX contains a key number.
; Search for an allocated block with that key number and return a pointer
; to that block (if found) in ES:DI. Return an error code (AX=3) if we
; cannot find the block.

TryShmAttach: cmp       al, 14h           ;Attach opcode.
              jne       IllegalOp
              mov       ax, SharedMemory
              mov       es, ax

              mov       di, ResidentSeg:AllocatedList
FindOurs:     cmp       dx, es:[di].Region.Key
              je        FoundOurs
              mov       di, es:[di].Region.Next
```

```
                test    di, di
                jne     FoundOurs
                mov     ax, 3           ;Can't find the key.
                iret

FoundOurs:      add     di, 8           ;Point at actual data.
                mov     ax, 0           ;No error.
                iret


; They called us with an illegal subfunction value. Try to do as little
; damage as possible.

IllegalOp:      mov     ax, 0         ;Who knows what they were thinking?
                iret
MyInt2F         endp
                assume  ds:nothing
ResidentSeg     ends


; Here's the segment that will actually hold the shared data.

SharedMemory    segment para public 'Shared'
                db      0FFFFh dup (?)
SharedMemory    ends




cseg            segment para public 'code'
                assume  cs:cseg, ds:ResidentSeg

; SeeIfPresent-         Checks to see if our TSR is already present in memory.
;                       Sets the zero flag if it is, clears the zero flag if
;                       it is not.

SeeIfPresent    proc    near
                push    es
                push    ds
                push    di
                mov     cx, 0ffh        ;Start with ID 0FFh.
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0           ;Verify presence call.
                int     2Fh
                pop     cx
                cmp     al, 0           ;Present in memory?
                je      TryNext
                strcmpl
                byte    "Dynamic Shared Memory TSR",0
                je      Success

TryNext:        dec     cl              ;Test USER IDs of 80h..FFh
                js      IDLoop
                cmp     cx, 0           ;Clear zero flag.
Success:        pop     di
                pop     ds
                pop     es
                ret
SeeIfPresent    endp



; FindID-       Determines the first (well, last actually) TSR ID available
;               in the multiplex interrupt chain. Returns this value in
;               the CL register.
;
;               Returns the zero flag set if it locates an empty slot.
;               Returns the zero flag clear if failure.

FindID          proc    near
                push    es
```

```
                push    ds
                push    di

                mov     cx, 0ffh        ;Start with ID 0FFh.
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0           ;Verify presence call.
                int     2Fh
                pop     cx
                cmp     al, 0           ;Present in memory?
                je      Success
                dec     cl              ;Test USER IDs of 80h..FFh
                js      IDLoop
                xor     cx, cx
                cmp     cx, 1           ;Clear zero flag
Success:        pop     di
                pop     ds
                pop     es
                ret
FindID          endp



Main            proc
                meminit

                mov     ax, ResidentSeg
                mov     ds, ax

                mov     ah, 62h         ;Get this program's PSP
                int     21h             ; value.
                mov     PSP, bx

; Before we do anything else, we need to check the command line
; parameters. If there is one, and it is the word "REMOVE", then remove
; the resident copy from memory using the multiplex (2Fh) interrupt.

                argc
                cmp     cx, 1           ;Must have 0 or 1 parms.
                jb      TstPresent
                je      DoRemove
Usage:          print
                byte    "Usage:",cr,lf
                byte    " shmalloc",cr,lf
                byte    "or shmalloc REMOVE",cr,lf,0
                ExitPgm


; Check for the REMOVE command.

DoRemove:       mov     ax, 1
                argv
                stricmpl
                byte    "REMOVE",0
                jne     Usage

                call    SeeIfPresent
                je      RemoveIt
                print
                byte    "TSR is not present in memory, cannot remove"
                byte    cr,lf,0
                ExitPgm

RemoveIt:       mov     MyTSRID, cl
                printf
                byte    "Removing TSR (ID #%d) from memory...",0
                dword   MyTSRID

                mov     ah, cl
                mov     al, 1           ;Remove cmd, ah contains ID
                int     2Fh
                cmp     al, 1           ;Succeed?
                je      RmvFailure
                print
```

```
                byte    “removed.”,cr,lf,0
                ExitPgm

RmvFailure:     print
                byte    cr,lf
                byte    “Could not remove TSR from memory.”,cr,lf
                byte    “Try removing other TSRs in the reverse order “
                byte    “you installed them.”,cr,lf,0
                ExitPgm



; Okay, see if the TSR is already in memory. If so, abort the
; installation process.

TstPresent:     call    SeeIfPresent
                jne     GetTSRID
                print
                byte    “TSR is already present in memory.”,cr,lf
                byte    “Aborting installation process”,cr,lf,0
                ExitPgm


; Get an ID for our TSR and save it away.

GetTSRID:       call    FindID
                je      GetFileName
                print
                byte    “Too many resident TSRs, cannot install”,cr,lf,0
                ExitPgm


; Things look cool so far, so install the interrupts

GetFileName:    mov     MyTSRID, cl
                print
                byte    “Installing interrupts...”,0


; Patch into the INT 2Fh interrupt chain.

                cli                     ;Turn off interrupts!
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[2Fh*4]
                mov     word ptr OldInt2F, ax
                mov     ax, es:[2Fh*4 + 2]
                mov     word ptr OldInt2F+2, ax
                mov     es:[2Fh*4], offset MyInt2F
                mov     es:[2Fh*4+2], seg ResidentSeg
                sti                     ;Okay, ints back on.

; We’re hooked up, the only thing that remains is to initialize the shared
; memory segment and then terminate and stay resident.

                printf
                byte    “Installed, TSR ID #%d.”,cr,lf,0
                dword   MyTSRID

                mov     ah, MyTSRID     ;Initialization call.
                mov     al, 13h
                int     2Fh

                mov     dx, EndResident ;Compute size of program.
                sub     dx, PSP
                mov     ax, 3100h       ;DOS TSR command.
                int     21h
Main            endp
cseg            ends

sseg            segment para stack ‘stack’
stk             db      256 dup (?)
sseg            ends
```

```
zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

We can modify the two applications from the previous section to try out this code:

```
; SHMAPP3.ASM
;
; This is a shared memory application that uses the dynamic shared memory
; TSR (SHMALLOC.ASM). This program inputs a string from the user and
; passes that string to SHMAPP4.ASM through the shared memory area.
;
;
                .xlist
                include     stdlib.a
                includelib stdlib.lib
                .list

dseg            segment     para public 'data'
ShmID           byte        0
dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent-Checks to see if the shared memory TSR is present in memory.
;               Sets the zero flag if it is, clears the zero flag if
;               it is not. This routine also returns the TSR ID in CL.

SeeIfPresent    proc        near
                push        es
                push        ds
                push        di
                mov         cx, 0ffh            ;Start with ID 0FFh.
IDLoop:         mov         ah, cl
                push        cx
                mov         al, 0               ;Verify presence call.
                int         2Fh
                pop         cx
                cmp         al, 0               ;Present in memory?
                je          TryNext
                strcmpl
                byte        "Dynamic Shared Memory TSR",0
                je          Success

TryNext:        dec         cl                  ;Test USER IDs of 80h..FFh
                js          IDLoop
                cmp         cx, 0               ;Clear zero flag.
Success:        pop         di
                pop         ds
                pop         es
                ret
SeeIfPresent    endp


; The main program for application #1 links with the shared memory
; TSR and then reads a string from the user (storing the string into
; shared memory) and then terminates.

Main            proc
                assume      cs:cseg, ds:dseg, es:SharedMemory
                mov         ax, dseg
                mov         ds, ax
                meminit
```

```
                print
                byte    "Shared memory application #3",cr,lf,0

; See if the shared memory TSR is around:

                call    SeeIfPresent
                je      ItsThere
                print
                byte    "Shared Memory TSR (SHMALLOC) is not loaded.",cr,lf
                byte    "This program cannot continue execution.",cr,lf,0
                ExitPgm


; Get the input line from the user:

ItsThere:       mov     ShmID, cl
                print
                byte    "Enter a string: ",0

                lea     di, InputLine   ;ES already points at proper seg.
                getsm

; The string is in our heap space. Let's move it over to the shared
; memory segment.

                strlen
                inc     cx              ;Add one for zero byte.
                push    es
                push    di

                mov     dx, 1234h       ;Our "key" value.
                mov     ah, ShmID
                mov     al, 11h         ;Shmalloc call.
                int     2Fh

                mov     si, di          ;Save as dest ptr.
                mov     dx, es

                pop     di              ;Retrive source address.
                pop     es
                strcpy                  ;Copy from local to shared.

                print
                byte    "Entered '",0
                puts
                print
                byte    "' into shared memory.",cr,lf,0


Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends

                end     Main




; SHMAPP4.ASM
;
; This is a shared memory application that uses the dynamic shared memory
; TSR (SHMALLOC.ASM). This program assumes the user has already run the
; SHMAPP3 program to insert a string into shared memory. This program
```

```
; simply prints that string from shared memory.
;
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

dseg            segment    para public 'data'
ShmID           byte       0
dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent-Checks to see if the shared memory TSR is present in memory.
;               Sets the zero flag if it is, clears the zero flag if
;               it is not. This routine also returns the TSR ID in CL.

SeeIfPresent    proc       near
                push       es
                push       ds
                push       di
                mov        cx, 0ffh          ;Start with ID 0FFh.
IDLoop:         mov        ah, cl
                push       cx
                mov        al, 0             ;Verify presence call.
                int        2Fh
                pop        cx
                cmp        al, 0             ;Present in memory?
                je         TryNext
                strcmpl
                byte       "Dynamic Shared Memory TSR",0
                je         Success

TryNext:        dec        cl                ;Test USER IDs of 80h..FFh
                js         IDLoop
                cmp        cx, 0             ;Clear zero flag.
Success:        pop        di
                pop        ds
                pop        es
                ret
SeeIfPresent    endp



; The main program for application #1 links with the shared memory
; TSR and then reads a string from the user (storing the string into
; shared memory) and then terminates.

Main            proc
                assume     cs:cseg, ds:dseg, es:SharedMemory
                mov        ax, dseg
                mov        ds, ax
                meminit

                print
                byte       "Shared memory application #4",cr,lf,0

; See if the shared memory TSR is around:

                call       SeeIfPresent
                je         ItsThere
                print
                byte       "Shared Memory TSR (SHMALLOC) is not loaded.",cr,lf
                byte       "This program cannot continue execution.",cr,lf,0
                ExitPgm

; If the shared memory TSR is present, get the address of the shared segment
; into the ES register:

ItsThere:       mov        ah, cl            ;ID of our TSR.
                mov        al, 14h           ;Attach call
                mov        dx, 1234h;Our "key" value
                int        2Fh
```

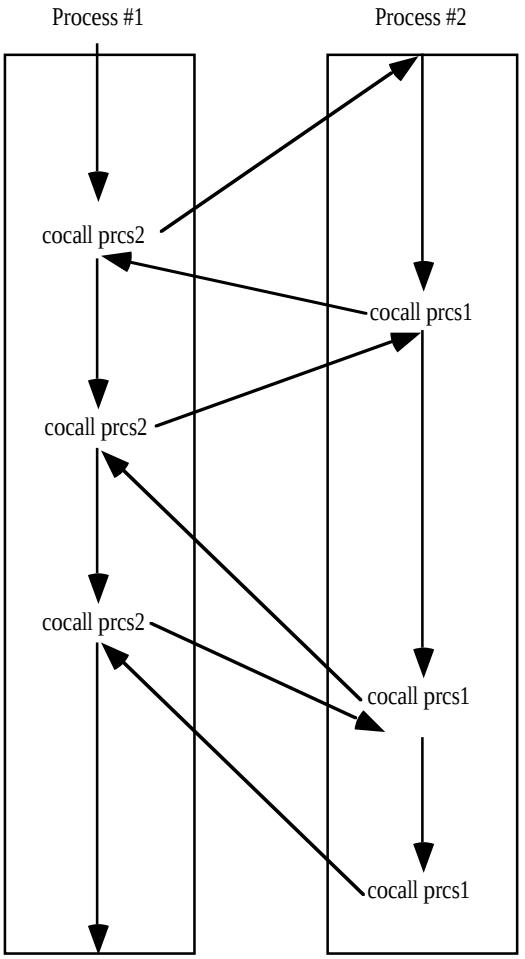
```
; Print the string input in SHMAPP3:

                print
                byte        “String from SHMAPP3 is ‘”,0

                puts

                print
                byte        “‘ from shared memory.”,cr,lf,0


Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg ends

sseg            segment     para stack ‘stack’
stk             db          1024 dup (“stack “)
sseg            ends

zzzzzzseg       segment     para public ‘zzzzzz’
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

19.3 Coroutines

DOS processes, even when using shared memory, suffer from one primary drawback – each program executes to completion before returning control back to the parent process. While this paradigm is suitable for many applications, it certainly does not suffice for all. A common paradigm is for two programs to swap control of the CPU back and forth while executing. This mechanism, slightly different from the subroutine call and return mechanism, is a *coroutine*.

Before discussing coroutines, it is probably a good idea to provide a solid definition for the term *process*. In a nutshell, a process is a program that is executing. A program can exist on the disk; processes exist in memory and have a program stack (with return addresses, etc.) associated with them. If there are multiple processes in memory at one time, each process must have its own program stack.

A *cocall* operation transfers control between two processes. A cocall is effectively a call and a return instruction all rolled into one operation. From the point of view of the process executing the cocall, the cocall operation is equivalent to a procedure call; from the point of view of the processing being called, the cocall operation is equivalent to a return operation. When the second process cocalls the first, control resumes *not at the beginning of the first process*, but immediately after the cocall operation. If two processes execute a sequence of mutual cocalls, control will transfer between the two processes in the following fashion:



Cocall Sequence Between Two Processes

Cocalls are quite useful for games where the “players” take turns, following different strategies. The first player executes some code to make its first move, then cocalls the second player and allows it to make a move. After the second player makes its move, it cocalls the first process and gives the first player its second move, picking up immediately after its cocall. This transfer of control bounces back and forth until one player wins.

The 80x86 CPUs do not provide a cocall instruction. However, it is easy to implement cocalls with existing instructions. Even so, there is little need for you to supply your own cocall mechanism, the UCR Standard Library provides a cocall package for 8086, 80186, and 80286 processors[2]. This package includes the `pcb` (process control block) data structure and three functions you can call: `coinit`, `cocall`, and `cocalll`.

The `pcb` structure maintains the current state of a process. The `pcb` maintains all the register values and other accounting information for a process. When a process makes a cocall, it stores the return address for the cocall in the `pcb`. Later, when some other process cocalls this process, the cocall operation simply reloads the registers, include `cs:ip`, from the `pcb` and that returns control to the next instruction after the first process' cocall. The `pcb` structure takes the following form:

```
pcb             struct
```

2. The cocall package works fine with the other processors as long as you don't use the 32-bit register set. Later, we will discuss how to extend the Standard Library routines to handle the 32-bit capabilities of the 80386 and late processors.

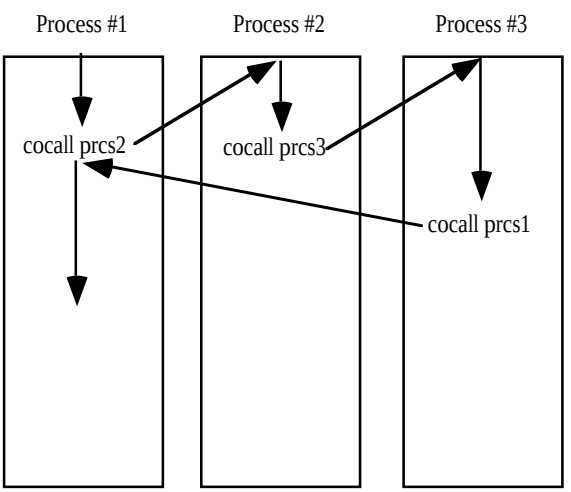
```
NextProc      dword     ?           ;Link to next PCB (for multitasking).
regsp         word      ?
regss         word      ?
regip         word      ?
regcs         word      ?
regax         word      ?
regbx         word      ?
regcx         word      ?
regdx         word      ?
regsi         word      ?
regdi         word      ?
regbp         word      ?
regds         word      ?
reges         word      ?
regflags      word      ?
PrcsID        word      ?
StartingTime  dword     ?           ;Used for multitasking accounting.
StartingDate  dword     ?           ;Used for multitasking accounting.
CPUTime       dword     ?           ;Used for multitasking accounting.
```

Four of these fields (as labelled) exist for preemptive multitasking and have no meaning for coroutines. We will discuss preemptive multitasking in the next section.

There are two important things that should be evident from this structure. First, the main reason the existing Standard Library coroutine support is limited to 16 bit register is because there is only room for the 16 bit versions of each of the registers in the pcb. If you want to support the 80386 and later 32 bit register sets, you would need to modify the pcb structure and the code that saves and restores registers in the pcb.

The second thing that should be evident is that the coroutine code preserves all registers across a cocall. This means you cannot pass information from one process to another in the registers when using a cocall. You will need to pass data between processes in global memory locations. Since coroutines generally exist in the same program, you will not even need to resort to the shared memory techniques. Any variables you declare in your data segment will be visible to all coroutines.

Note, by the way, that a program may contain more than two coroutines. If coroutine one cocalls coroutine two, and coroutine two cocalls coroutine three, and then coroutine three cocalls coroutine one, coroutine one picks up immediately after the cocall it made to coroutine two.



Cocalls Between Three Processes

Since a cocall effectively *returns* to the target coroutine, you might wonder what happens on the *first* cocall to any process. After all, if that process has not executed any code, there is no "return address" where you can resume execution. This is an easy problem to solve, we need only initialize the return address of such a process to the address of the first instruction to execute in that process.

A similar problem exists for the stack. When a program begins execution, the main program (coroutine one) takes control and uses the stack associated with the entire program. Since each process must have its own stack, where do the other coroutines get their stacks?

The easiest way to initialize the stack and initial address for a coroutine is to do this when declaring a pcb for a process. Consider the following pcb variable declaration:

```
ProcessTwo     pcb        {0,           offset EndStack2, seg EndStack2,
                                        offset StartLoc2, seg StartLoc2}
```

This definition initializes the NextProc field with NULL (the Standard Library coroutine functions do not use this field) and initialize the ss:sp and cs:ip fields with the last address of a stack area (EndStack2) and the first instruction of the process (StartLoc2). Now all you need to do is reserve a reasonable amount of stack storage for the process. You can create multiple stacks in the SHELL.ASM sseg as follows:

```
sseg           segment    para stack 'stack'

; Stack for process #2:

stk2           byte       1024 dup (?)
EndStack2      word       ?

; Stack for process #3:

stk3           byte       1024 dup (?)
EndStack3      word       ?

; The primary stack for the main program (process #1) must appear at
; the end of sseg.

stk            byte       1024 dup (?)
sseg           ends
```

There is the question of "how much space should one reserve for each stack?" This, of course, varies with the application. If you have a simple application that doesn't use recursion or allocate any local variables on the stack, you could get by with as little as 256 bytes of stack space for a process. On the other hand, if you have recursive routines or allocate storage on the stack, you will need considerably more space. For simple programs, 1-8K stack storage should be sufficient. Keep in mind that you can allocate a maximum of 64K in the SHELL.ASM sseg. If you need additional stack space, you will need to up the other stacks in a different segment (they do not need to be in sseg, it's just a convenient place for them) or you will need to allocate the stack space differently.

Note that you do not have to allocate the stack space as an array within your program. You can also allocate stack space dynamically using the Standard Library malloc call. The following code demonstrates how to set up an 8K dynamically allocated stack for the pcb variable Process2:

```
mov        cx, 8192
malloc
jc         InsufficientRoom
mov        Process2.ss, es
mov        Process2.sp, di
```

Setting up the coroutines the main program will call is pretty easy. However, there is the issue of setting up the pcb for the main program. You cannot initialize the pcb for the main program the same way you initialize the pcb for the other processes; it is already running and has valid cs:ip and ss:sp values. Were you to initialize the main program's pcb the same way we did for the other processes, the system would simply restart the main program when you make a cocall back to it. To initialize the pcb for the main program, you must use the coinit function. The coinit function expects you to pass it the address of the main program's pcb in the es:di register pair. It initializes some variables internal to the Standard Library so the first cocall operation will save the 80x86 machine state in the pcb you specify by es:di. After the coinit call, you can begin making cocalls to other processes in your program.

To cocall a coroutine, you use the Standard Library cocall function. The cocall function call takes two forms. Without any parameters this function transfers control to the coroutine whose pcb address appears in the es:di register pair. If the address of a pcb appears in the operand field of this instruction, cocall transfers control to the specified coroutine (don't forget, the name of the pcb, *not* the process, must appear in the operand field).

The best way to learn how to use coroutines is via example. The following program is an interesting piece of code that generates mazes on the PC's display. The maze generation algorithm has one major constraint – there must be no more than one correct solution to the maze (it is possible for there to be no solution). The main program creates a set of background processes called "demons" (actually, daemon is the correct term, but demon sounds more appropriate here). Each demon begins carving out a portion of the maze subject to the main constraint. Each demon gets to dig one cell from the maze and then it passes control to another demon. As it turns out, demons can "dig themselves into a corner" and die (demons live only to dig). When this happens, the demon removes itself from the list of active demons. When all demons die off, the maze is (in theory) complete. Since the demons die off fairly regularly, there must be some mechanism to create new demons. Therefore, this program randomly spawns new demons who start digging their own tunnels perpendicular to their parents. This helps ensure that there is a sufficient supply of demons to dig out the entire maze; the demons all die off only when there are no, or few, cells remaining to dig in the maze.

```
; AMAZE.ASM
;
; A maze generation/solution program.
;
; This program generates an 80x25 maze and directly draws the maze on the
; video display. It demonstrates the use of coroutines within a program.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

byp             textequ    <byte ptr>

dseg            segment    para public 'data'

; Constants:
;
; Define the "ToScreen" symbol (to any value) if the maze is 80x25 and you
; want to display it on the video screen.

ToScreen        equ        0


; Maximum X and Y coordinates for the maze (matching the display).

MaxXCoord       equ        80
MaxYCoord       equ        25

; Useful X,Y constants:

WordsPerRow     =          MaxXCoord+2
BytesPerRow     =          WordsPerRow*2

StartX          equ        1                ;Starting X coordinate for maze
StartY          equ        3                ;Starting Y coordinate for maze
EndX            equ        MaxXCoord        ;Ending X coordinate for maze
EndY            equ        MaxYCoord-1      ;Ending Y coordinate for maze

EndLoc          =          ( (EndY-1)*MaxXCoord + EndX-1)*2
StartLoc        =          ( (StartY-1)*MaxXCoord + StartX-1)*2

; Special 16-bit PC character codes for the screen for symbols drawn during
; maze generation. See the chapter on the video display for details.

                ifdef      mono             ;Mono display adapter.

WallChar        equ        7dbh             ;Solid block character
```

```
NoWallChar      equ         720h                ;space
VisitChar       equ         72eh                ;Period
PathChar        equ         72ah                ;Asterisk

                else                            ;Color display adapter.

WallChar        equ         1dbh                ;Solid block character
NoWallChar      equ         0edbh               ;space
VisitChar       equ         0bdbh               ;Period
PathChar        equ         4e2ah               ;Asterisk

                endif



; The following are the constants that may appear in the Maze array:

Wall            =           0
NoWall          =           1
Visited         =           2

; The following are the directions the demons can go in the maze

North           =           0
South           =           1
East            =           2
West            =           3


; Some important variables:


; The Maze array must contain an extra row and column around the
; outside edges for our algorithm to work properly.

Maze            word        (MaxYCoord+2) dup ((MaxXCoord+2) dup (Wall))

; The follow macro computes an index into the above array assuming
; a demon's X and Y coordinates are in the dl and dh registers, respectively.
; Returns index in the AX register

MazeAdrs        macro
                mov         al, dh
                mov         ah, WordsPerRow     ;Index into array is computed
                mul         ah                  ; by (Y*words/row + X)*2.
                add         al, dl
                adc         ah, 0
                shl         ax, 1               ;Convert to byte index
                endm

; The following macro computes an index into the screen array, using the
; same assumptions as above. Note that the screen matrix is 80x25 whereas
; the maze matrix is 82x27; The X/Y coordinates in DL/DH are 1..80 and
; 1..25 rather than 0..79 and 0..24 (like we need). This macro adjusts
; for that.

ScrnAdrs        macro
                mov         al, dh
                dec         al
                mov         ah, MaxXCoord
                mul         ah
                add         al, dl
                adc         ah, 0
                dec         ax
                shl         ax, 1
                endm



; PCB for the main program. The last live demon will call this guy when
; it dies.

MainPCB         pcb         {}
```

```
; List of up to 32 demons.

MaxDemons     =         32              ;Must be a power of two.
ModDemons     =         MaxDemons-1     ;Mask for MOD computation.

DemonList     pcb       MaxDemons dup ({})

DemonIndex    byte      0               ;Index into demon list.
DemonCnt      byte      0               ;Number of demons in list.


; Random number generator seed (we'll use our random number generator
; rather than the standard library's because we want to be able to specify
; an initial seed value).

Seed          word      0

dseg          ends



; The following is the segment address of the video display, change this
; from 0B800h to 0B000h if you have a monochrome display rather than a
; color display.

ScreenSeg     segment   at 0b800h
Screen        equ       this word       ;Don't generate in date here!
ScreenSeg     ends


cseg          segment   para public 'code'
              assume    cs:cseg, ds:dseg

; Totally bogus random number generator, but we don't need a really
; great one for this program. This code uses its own random number
; generator rather than the one in the Standard Library so we can
; allow the user to use a fixed seed to produce the same maze (with
; the same seed) or different mazes (by choosing different seeds).

RandNum       proc      near
              push      cx
              mov       cl, byte ptr Seed
              and       cl, 7
              add       cl, 4
              mov       ax, Seed
              xor       ax, 55aah
              rol       ax, cl
              xor       ax, Seed
              inc       ax
              mov       Seed, ax
              pop       cx
              ret
RandNum       endp

; Init-  Handles all the initialization chores for the main program.
;        In particular, it initializes the coroutine package, gets a
;        random number seed from the user, and initializes the video display.

Init          proc      near
              print
              byte      "Enter a small integer for a random number seed:",0
              getsm
              atoi
              free
              mov       Seed, ax

; Fill the interior of the maze with wall characters, fill the outside
; two rows and columns with nowall values. This will prevent the demons
; from wandering outside the maze.


; Fill the first row with Visited values.
```

```
                cld
                mov     cx, WordsPerRow
                lesi    Maze
                mov     ax, Visited
        rep     stosw

; Fill the last row with NoWall values.

                mov     cx, WordsPerRow
                lea     di, Maze+(MaxYCoord+1)*BytesPerRow
        rep     stosw

; Write a NoWall value to the starting position:

                mov     Maze+(StartY*WordsPerRow+StartX)*2, NoWall


; Write NoWall values along the two vertical edges of the maze.

                lesi    Maze
                mov     cx, MaxYCoord+1
EdgesLoop:      mov     es:[di], ax                 ;Plug the left edge.
                mov     es:[di+BytesPerRow-2], ax   ;Plug the right edge.
                add     di, BytesPerRow
                loop    EdgesLoop


                ifdef   ToScreen

; Okay, fill the screen with WallChar values:

                lesi    Screen
                mov     ax, WallChar
                mov     cx, 2000
        rep     stosw

; Write appropriate characters to the starting and ending locations:

                mov     word ptr es:Screen+EndLoc, PathChar
                mov     word ptr es:Screen+StartLoc, NoWallChar

                endif                   ;ToScreen


; Zero out the DemonList:

                mov     cx, (size pcb)*MaxDemons
                lea     di, DemonList
                mov     ax, dseg
                mov     es, ax
                xor     ax, ax
        rep     stosb

                ret
Init            endp



; CanStart- This function checks around the current position
; to see if the maze generator can start digging a new tunnel
; in a direction perpendicular to the current tunnel. You can
; only start a new tunnel if there are wall characters for at
; least two positions in the desired direction:
;
;                        ##
;                       *##
;                        ##
;
; If "*" is current position and "#" represent wall characters
; and the current direction is north or south, then it is okay
; for the maze generator to start a new path in the east dir-
; ection. Assuming "." represents a tunnel, you cannot start
; a new tunnel in the east direction if any of the following
; patterns occur:
```

```
;
;        .#   #.           ##        ##              ##     ##
;       *##  *##          *.#       *#.             *##    *##
;        ##   ##           ##        ##              .#     #.
;
; CanStart returns true (carry set) if we can start a new tunnel off the
; path being dug by the current demon.
;
; On entry,    dl is demon's X-Coordinate
;              dh is demon's Y-Coordinate
;              cl is demon's direction

CanStart      proc      near
              push      ax
              push      bx

              MazeAdrs                ;Compute index to demon(x,y) in maze.
              mov       bx, ax

; CL contains the current direction, 0=north, 1=south, 2=east, 3=west.
; Note that we can test bit #1 for north/south (0) or east/west (1).

              test      cl, 10b       ;See if north/south or east/west
              jz        NorthSouth

; If the demon is going in an east or west direction, we can start a new
; tunnel if there are six wall blocks just above or below the current demon.
; Note: We are checking if all values in these six blocks are Wall values.
; This code depends on the fact that Wall characters are zero and the sum
; of these six blocks will be zero if a move is possible.

              mov       al, byp Maze[bx+BytesPerRow*2] ;Maze[x, y+2]
              add       al, byp Maze[bx+BytesPerRow*2+2] ;Maze[x+1,y+2]
              add       al, byp Maze[bx+BytesPerRow*2-2] ;Maze[x-1,y+2]
              je        ReturnTrue

              mov       al, byp Maze[bx-BytesPerRow*2] ;Maze[x, y-2]
              add       al, byp Maze[bx-BytesPerRow*2+2] ;Maze[x+1,y-2]
              add       al, byp Maze[bx-BytesPerRow*2-2] ;Maze[x-1,y-2]
              je        ReturnTrue

ReturnFalse:  clc                     ;Clear carry = false.
              pop       bx
              pop       ax
              ret

; If the demon is going in a north or south direction, we can start a
; new tunnel if there are six wall blocks just to the left or right
; of the current demon.

NorthSouth:   mov       al, byp Maze[bx+4];Maze[x+2,y]
              add       al, byp Maze[bx+BytesPerRow+4];Maze[x+2,y+1]
              add       al, byp Maze[bx-BytesPerRow+4];Maze[x+2,y-1]
              je        ReturnTrue

              mov       al, byp Maze[bx-4];Maze[x-2,y]
              add       al, byp Maze[bx+BytesPerRow-4];Maze[x-2,y+1]
              add       al, byp Maze[bx-BytesPerRow-4];Maze[x-2,y-1]
              jne       ReturnFalse

ReturnTrue:   stc                     ;Set carry = true.
              pop       bx
              pop       ax
              ret
CanStart      endp



; CanMove-     Tests to see if the current demon (dir=cl, x=dl, y=dh) can
;              move in the specified direction. Movement is possible if
;              the demon will not come within one square of another tunnel.
;              This function returns true (carry set) if a move is possible.
;              On entry, CH contains the direction this code should test.
```

```
CanMove         proc
                push        ax
                push        bx

                MazeAdrs                        ;Put @Maze[x,y] into ax.
                mov         bx, ax

                cmp         ch, South
                jb          IsNorth
                je          IsSouth
                cmp         ch, East
                je          IsEast

; If the demon is moving west, check the blocks in the rectangle formed
; by Maze[x-2,y-1] to Maze[x-1,y+1] to make sure they are all wall values.

                mov         al, byp Maze[bx-BytesPerRow-4];Maze[x-2, y-1]
                add         al, byp Maze[bx-BytesPerRow-2];Maze[x-1, y-1]
                add         al, byp Maze[bx-4];Maze[x-2, y]
                add         al, byp Maze[bx-2];Maze[x-1, y]
                add         al, byp Maze[bx+BytesPerRow-4];Maze[x-2, y+1]
                add         al, byp Maze[bx+BytesPerRow-2];Maze[x-1, y+1]
                je          ReturnTrue
ReturnFalse:    clc
                pop         bx
                pop         ax
                ret


; If the demon is going east, check the blocks in the rectangle formed
; by Maze[x+1,y-1] to Maze[x+2,y+1] to make sure they are all wall values.

IsEast:         mov         al, byp Maze[bx-BytesPerRow+4];Maze[x+2, y-1]
                add         al, byp Maze[bx-BytesPerRow+2];Maze[x+1, y-1]
                add         al, byp Maze[bx+4];Maze[x+2, y]
                add         al, byp Maze[bx+2];Maze[x+1, y]
                add         al, byp Maze[bx+BytesPerRow+4];Maze[x+2, y+1]
                add         al, byp Maze[bx+BytesPerRow+2];Maze[x+1, y+1]
                jne         ReturnFalse
ReturnTrue:     stc
                pop         bx
                pop         ax
                ret


; If the demon is going north, check the blocks in the rectangle formed
; by Maze[x-1,y-2] to Maze[x+1,y-1] to make sure they are all wall values.

IsNorth:        mov         al, byp Maze[bx-BytesPerRow-2];Maze[x-1, y-1]
                add         al, byp Maze[bx-BytesPerRow*2-2];Maze[x-1, y-2]
                add         al, byp Maze[bx-BytesPerRow];Maze[x, y-1]
                add         al, byp Maze[bx-BytesPerRow*2];Maze[x, y-2]
                add         al, byp Maze[bx-BytesPerRow+2];Maze[x+1, y-1]
                add         al, byp Maze[bx-BytesPerRow*2+2];Maze[x+1, y-2]
                jne         ReturnFalse
                stc
                pop         bx
                pop         ax
                ret


; If the demon is going south, check the blocks in the rectangle formed
; by Maze[x-1,y+2] to Maze[x+1,y+1] to make sure they are all wall values.

IsSouth:        mov         al, byp Maze[bx+BytesPerRow-2];Maze[x-1, y+1]
                add         al, byp Maze[bx+BytesPerRow*2-2];Maze[x-1, y+2]
                add         al, byp Maze[bx+BytesPerRow];Maze[x, y+1]
                add         al, byp Maze[bx+BytesPerRow*2];Maze[x, y+2]
                add         al, byp Maze[bx+BytesPerRow+2];Maze[x+1, y+1]
                add         al, byp Maze[bx+BytesPerRow*2+2];Maze[x+1, y+2]
                jne         ReturnFalse
                stc
```

```
                pop     bx
                pop     ax
                ret

CanMove         endp



; SetDir- Changes the current direction. The maze digging algorithm has
; decided to change the direction of the tunnel begin dug by one
; of the demons. This code checks to see if we CAN change the direction,
; and picks a new direction if possible.
;
; If the demon is going north or south, a direction change causes the demon
; to go east or west. Likewise, if the demon is going east or west, a
; direction change forces it to go north or south. If the demon cannot
; change directions (because it cannot move in the new direction for one
; reason or another), SetDir returns without doing anything. If a direction
; change is possible, then SetDir selects a new direction. If there is only
; one possible new direction, the demon is sent off in that direction.
; If the demon could move off in one of two different directions, SetDir
; “flips a coin” to choose one of the two new directions.
;
; This function returns the new direction in al.

SetDir          proc    near

                test    cl, 10b         ;See if north/south
                je      IsNS            ; or east/west direction.

; We’re going east or west. If we can move EITHER north or south from
; this point, randomly choose one of the directions. If we can only
; move one way or the other, choose that direction. If we can’t go either
; way, return without changing the direction.

                mov     ch, North       ;See if we can move north
                call    CanMove
                jnc     NotNorth
                mov     ch, South       ;See if we can move south
                call    CanMove
                jnc     DoNorth
                call    RandNum         ;Get a random direction
                and     ax, 1           ;Make it north or south.
                ret

DoNorth:        mov     ax, North
                ret

NotNorth:       mov     ch, South
                call    CanMove
                jnc     TryReverse
DoSouth:        mov     ax, South
                ret



; If the demon is moving north or south, choose a new direction of east
; or west, if possible.

IsNS:           mov     ch, East        ;See if we can move East
                call    CanMove
                jnc     NotEast
                mov     ch, West        ;See if we can move West
                call    CanMove
                jnc     DoEast
                call    RandNum         ;Get a random direction
                and     ax, 1b          ;Make it East or West
                or      al, 10b
                ret

DoEast:         mov     ax, East
                ret
```

```
DoWest:         mov         ax, West
                ret

NotEast:        mov         ch, West
                call        CanMove
                jc          DoWest

; Gee, we can't switch to a perpendicular direction, see if we can
; turn around.

TryReverse:     mov         ch, cl
                xor         ch, 1
                call        CanMove
                jc          ReverseDir

; If we can't turn around (likely), then keep going in the same direction.

                mov         ah, 0
                mov         al, cl              ;Stay in same direction.
                ret

; Otherwise reverse direction down here.

ReverseDir:     mov         ah, 0
                mov         al, cl
                xor         al, 1
                ret
SetDir          endp



; Stuck-        This function checks to see if a demon is stuck and cannot
;               move in any direction. It returns true if the demon is
;               stuck and needs to be killed.

Stuck           proc        near
                mov         ch, North
                call        CanMove
                jc          NotStuck
                mov         ch, South
                call        CanMove
                jc          NotStuck
                mov         ch, East
                call        CanMove
                jc          NotStuck
                mov         ch, West
                call        CanMove
NotStuck:       ret
Stuck           endp



; NextDemon-    Searches through the demon list to find the next available
;               active demon. Return a pointer to this guy in es:di.

NextDemon       proc        near
                push        ax

NDLoop:         inc         DemonIndex          ;Move on to next demon,
                and         DemonIndex, ModDemons ; MOD MaxDemons.
                mov         al, size pcb        ;Compute index into
                mul         DemonIndex          ; DemonList.
                mov         di, ax              ;See if the demon at this
                add         di, offset DemonList ; offset is active.
                cmp         byp [di].pcb.NextProc, 0
                je          NDLoop

                mov         ax, ds
                mov         es, ax
                pop         ax
                ret
NextDemon       endp
```

```
; Dig-         This is the demon process.
;              It moves the demon one position (if possible) in its current
;              direction. After moving one position forward, there is
;              a 25% chance that this guy will change its direction; there
;              is a 25% chance this demon will spawn a child process to
;              dig off in a perpendicular direction.

Dig            proc       near

; See if the current demon is stuck. If the demon is stuck, then we've
; go to remove it from the demon list. If it is not stuck, then have it
; continue digging. If it is stuck and this is the last active demon,
; then return control to the main program.

               call       Stuck
               jc         NotStuck

; Okay, kill the current demon.
; Note: this will never kill the last demon because we have the timer
; process running. The timer process is the one that always stops
; the program.

               dec        DemonCnt

; Since the count is not zero, there must be more demons in the demon
; list. Free the stack space associated with the current demon and
; then search out the next active demon and have at it.

MoreDemons:    mov        al, size pcb
               mul        DemonIndex
               mov        bx, ax

; Free the stack space associated with this process. Note this code is
; naughty. It assumes the stack is allocated with the Standard Library
; malloc routine that always produces a base address of 8.

               mov        es, DemonList[bx].regss
               mov        di, 8                           ;Cheating!
               free

; Mark the demon entry for this guy as unused.

               mov        byp DemonList[bx].NextProc, 0   ;Mark as unused.


; Okay, locate the next active demon in the list.

FndNxtDmn:     call       NextDemon
               cocall                      ;Never returns




; If the demon is not stuck, then continue digging away.

NotStuck:      mov        ch, cl
               call       CanMove
               jnc        DontMove

; If we can move, then adjust the demon's coordinates appropriately:

               cmp        cl, South
               jb         MoveNorth
               je         MoveSouth
               cmp        cl, East
               jne        MoveWest

; Moving East:

               inc        dl
               jmp        MoveDone

MoveWest:      dec        dl
```

```
                jmp         MoveDone

MoveNorth:      dec         dh
                jmp         MoveDone

MoveSouth:inc   dh

; Okay, store a NoWall value at this entry in the maze and output a NoWall
; character to the screen (if writing data to the screen).

MoveDone:       MazeAdrs
                mov         bx, ax
                mov         Maze[bx], NoWall

                ifdef       ToScreen
                ScrnAdrs
                mov         bx, ax
                push        es
                mov         ax, ScreenSeg
                mov         es, ax
                mov         word ptr es:[bx], NoWallChar
                pop         es
                endif

; Before leaving, see if this demon shouldn't change direction.

DontMove:       call        RandNum
                and         al, 11b             ;25% chance result is zero.
                jne         NoChangeDir
                call        SetDir
                mov         cl, al

NoChangeDir:


; Also, see if this demon should spawn a child process

                call        RandNum
                and         al, 11b             ;Give it a 25% chance.
                jne         NoSpawn

; Okay, see if it's possible to spawn a new process at this point:

                call        CanStart
                jnc         NoSpawn

; See if we've already got MaxDemons active:

                cmp         DemonCnt, MaxDemons
                jae         NoSpawn

                inc         DemonCnt            ;Add another demon.


; Okay, create a new demon and add him to the list.

                push        dx                  ;Save cur demon info.
                push        cx

; Locate a free slot for this demon

                lea         si, DemonList- size pcb
FindSlot:       add         si, size pcb
                cmp         byp [si].pcb.NextProc, 0
                jne         FindSlot


; Allocate some stack space for the new demon.

                mov         cx, 256             ;256 byte stack.
                malloc

; Set up the stack pointer for this guy:
```

```
                add     di, 248         ;Point stack at end.
                mov     [si].pcb.regss, es
                mov     [si].pcb.regsp, di

; Set up the execution address for this guy:

                mov     [si].pcb.regcs, cs
                mov     [si].pcb.regip, offset Dig

; Initial coordinates and direction for this guy:

                mov     [si].pcb.regdx, dx

; Select a direction for this guy.

                pop     cx              ;Retrieve direction.
                push    cx

                call    SetDir
                mov     ah, 0
                mov     [si].pcb.regcx, ax

; Set up other misc junk:

                mov     [si].pcb.regds, seg dseg
                sti
                pushf
                pop     [si].pcb.regflags
                mov     byp [si].pcb.NextProc, 1        ;Mark active.


; Restore current process' parameters

                pop     cx              ;Restore current demon.
                pop     dx

NoSpawn:

; Okay, with all of the above done, it's time to pass control on to a new
; digger. The following cocall passes control to the next digger in the
; DemonList.

GetNextDmn:     call    NextDemon

; Okay, we've got a pointer to the next demon in the list (might be the
; same demon if there's only one), pass control to that demon.

                cocall
                jmp     Dig
Dig             endp


; TimerDemon-   This demon introduces a delay between
;               each cycle in the demon list. This slows down the
;               maze generation so you can see the maze being built
;               (which makes the program more interesting to watch).

TimerDemon      proc    near
                push    es
                push    ax

                mov     ax, 40h         ;BIOS variable area
                mov     es, ax
                mov     ax, es:[6Ch]    ;BIOS timer location
Wait4Change:    cmp     ax, es:[6Ch]    ;BIOS changes this every
                je      Wait4Change     ; 1/18th second.

                cmp     DemonCnt, 1
                je      QuitProgram
                pop     es
                pop     ax
                call    NextDemon
                cocall
                jmp     TimerDemon
```

```
QuitProgram:  cocall    MainPCB          ;Quit the program
TimerDemon    endp



; What good is a maze generator program if it cannot solve the mazes it
; creates? SolveMaze finds the solution (if any) for this maze. It marks
; the solution path and the paths it tried, but failed on.
;
; function solvemaze(x,y:integer):boolean

sm_X          textequ   <[bp+6]>
sm_Y          textequ   <[bp+4]>

SolveMaze     proc      near
              push      bp
              mov       bp, sp

; See if we've just solved the maze:

              cmp       byte ptr sm_X, EndX
              jne       NotSolved
              cmp       byte ptr sm_Y, EndY
              jne       NotSolved
              mov       ax, 1            ;Return true.
              pop       bp
              ret       4

; See if moving to this spot was an illegal move. There will be
; a NoWall value at this cell in the maze if the move is legal.

NotSolved:    mov       dl, sm_X
              mov       dh, sm_Y
              MazeAdrs
              mov       bx, ax
              cmp       Maze[bx], NoWall
              je        MoveOK
              mov       ax, 0            ;Return failure
              pop       bp
              ret       4

; Well, it is possible to move to this point, so place an appropriate
; value on the screen and keep searching for the solution.

MoveOK:       mov       Maze[bx], Visited

              ifdef     ToScreen
              push      es               ;Write a "VisitChar"
              ScrnAdrs                   ; character to the
              mov       bx, ax           ; screen at this X,Y
              mov       ax, ScreenSeg    ; position.
              mov       es, ax
              mov       word ptr es:[bx], VisitChar
              pop       es
              endif

; Recusively call SolveMaze until we get a solution. Just call SolveMaze
; for the four possible directions (up, down, left, right) we could go.
; Since we've left "Visited" values in the Maze, we will not accidentally
; search back through the path we've already travelled. Furthermore, if
; we cannot go in one of the four directions, SolveMaze will catch this
; immediately upon entry (see the code at the start of this routine).

              mov       ax, sm_X         ;Try the path at location
              dec       ax               ; (X-1, Y)
              push      ax
              push      sm_Y
              call      SolveMaze
              test      ax, ax           ;Solution?
              jne       Solved

              push      sm_X             ;Try the path at location
```

```
                mov     ax, sm_Y        ; (X, Y-1)
                dec     ax
                push    ax
                call    SolveMaze
                test    ax, ax          ;Solution?
                jne     Solved

                mov     ax, sm_X        ;Try the path at location
                inc     ax              ; (X+1, Y)
                push    ax
                push    sm_Y
                call    SolveMaze
                test    ax, ax          ;Solution?
                jne     Solved

                push    sm_X            ;Try the path at location
                mov     ax, sm_Y        ; (X, Y+1)
                inc     ax
                push    ax
                call    SolveMaze
                test    ax, ax          ;Solution?
                jne     Solved
                pop     bp
                ret     4

Solved:
                ifdef   ToScreen        ;Draw return path.
                push    es
                mov     dl, sm_X
                mov     dh, sm_Y
                ScrnAdrs
                mov     bx, ax
                mov     ax, ScreenSeg
                mov     es, ax
                mov     word ptr es:[bx], PathChar
                pop     es
                mov     ax, 1           ;Return true
                endif

                pop     bp
                ret     4
SolveMaze       endp


; Here's the main program that drives the whole thing:

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit


                call    Init            ;Initialize maze stuff.
                lesi    MainPCB         ;Initialize coroutine
                coinit                  ; package.

; Create the first demon.
; Set up the stack pointer for this guy:

                mov     cx, 256
                malloc
                add     di, 248
                mov     DemonList.regsp, di
                mov     DemonList.regss, es

; Set up the execution address for this guy:

                mov     DemonList.regcs, cs
                mov     DemonList.regip, offset Dig

; Initial coordinates and direction for this guy:
```

```
                mov     cx, East          ;Start off going east.
                mov     dh, StartY
                mov     dl, StartX
                mov     DemonList.regcx, cx
                mov     DemonList.regdx, dx

; Set up other misc junk:

                mov     DemonList.regds, seg dseg
                sti
                pushf
                pop     DemonList.regflags
                mov     byp DemonList.NextProc, 1       ;Demon is “active”.
                inc     DemonCnt
                mov     DemonIndex, 0



; Set up the Timer demon:

                mov     DemonList.regsp+(size pcb), offset EndTimerStk
                mov     DemonList.regss+(size pcb), ss

; Set up the execution address for this guy:

                mov     DemonList.regcs+(size pcb), cs
                mov     DemonList.regip+(size pcb), offset TimerDemon

; Set up other misc junk:

                mov     DemonList.regds+(size pcb), seg dseg
                sti
                pushf
                pop     DemonList.regflags+(size pcb)
                mov     byp DemonList.NextProc+(size pcb), 1
                inc     DemonCnt

; Start the ball rolling.

                mov     ax, ds
                mov     es, ax
                lea     di, DemonList
                cocall

; Wait for the user to press a key before solving the maze:

                getc

                mov     ax, StartX
                push    ax
                mov     ax, StartY
                push    ax
                call    SolveMaze

; Wait for another keystroke before quitting:

                getc

                mov     ax, 3             ;Clear screen and reset video mode.
                int     10h

Quit:           ExitPgm                   ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment   para stack ‘stack’

; Stack for the timer demon we create (we’ll allocate the other
; stacks dynamically).

TimerStk        byte      256 dup (?)
EndTimerStk     word      ?
```

```
; Main program's stack:

stk             byte        512 dup (?)
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

The existing Standard Library coroutine package is not suitable for programs that use the 80386 and later 32 bit register sets. As mentioned earlier, the problem lies in the fact that the Standard Library only preserves the 16-bit registers when switching between processes. However, it is a relatively trivial extension to modify the Standard Library so that it saves 32 bit registers. To do so, just change the definition of the pcb (to make room for the 32 bit registers) and the sl_cocall routine:

```
                .386
                option      segment:use16

dseg            segment     para public 'data'

wp              equ         <word ptr>

; 32-bit PCB. Note we only keep the L.O. 16 bits of SP since we are
; operating in real mode.

pcb32           struc
regsp           word        ?
regss           word        ?
regip           word        ?
regcs           word        ?

regeax          dword       ?
regebx          dword       ?
regecx          dword       ?
regedx          dword       ?
regesi          dword       ?
regedi          dword       ?
regebp          dword       ?

regds           word        ?
reges           word        ?
regflags        dword       ?
pcb32           ends


DefaultPCB      pcb32       <>
DefaultCortn    pcb32       <>

CurCoroutine    dword       DefaultCortn    ;Points at the currently executing
                                            ; coroutine.
dseg            ends


cseg            segment     para public 'slcode'

;===============================================================================
;
; 32-Bit Coroutine support.
;
; COINIT32- ES:DI contains the address of the current (default) process' PCB.

CoInit32        proc        far
                assume      ds:dseg
                push        ax
```

```
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     wp dseg:CurCoroutine, di
                mov     wp dseg:CurCoroutine+2, es
                pop     ds
                pop     ax
                ret
CoInit32        endp


; COCALL32- transfers control to a coroutine. ES:DI contains the address
; of the PCB. This routine transfers control to that coroutine and then
; returns a pointer to the caller's PCB in ES:DI.

cocall32        proc    far
                assume  ds:dseg
                pushfd
                push    ds
                push    es              ;Save these for later
                push    edi
                push    eax
                mov     ax, dseg
                mov     ds, ax
                cli                     ;Critical region ahead.


; Save the current process' state:

                les     di, dseg:CurCoroutine
                pop     es:[di].pcb32.regeax
                mov     es:[di].pcb32.regebx, ebx
                mov     es:[di].pcb32.regecx, ecx
                mov     es:[di].pcb32.regedx, edx
                mov     es:[di].pcb32.regesi, esi
                pop     es:[di].pcb32.regedi
                mov     es:[di].pcb32.regebp, ebp

                pop     es:[di].pcb32.reges
                pop     es:[di].pcb32.regds
                pop     es:[di].pcb32.regflags
                pop     es:[di].pcb32.regip
                pop     es:[di].pcb32.regcs
                mov     es:[di].pcb32.regsp, sp
                mov     es:[di].pcb32.regss, ss

                mov     bx, es          ;Save so we can return in
                mov     ecx, edi      ; ES:DI later.
                mov     edx, es:[di].pcb32.regedi
                mov     es, es:[di].pcb32.reges
                mov     di, dx          ;Point es:di at new PCB

                mov     wp dseg:CurCoroutine, di
                mov     wp dseg:CurCoroutine+2, es

                mov     es:[di].pcb32.regedi, ecx ;The ES:DI return values.
                mov     es:[di].pcb32.reges, bx

; Okay, switch to the new process:

                mov     ss, es:[di].pcb32.regss
                mov     sp, es:[di].pcb32.regsp
                mov     eax, es:[di].pcb32.regeax
                mov     ebx, es:[di].pcb32.regebx
                mov     ecx, es:[di].pcb32.regecx
                mov     edx, es:[di].pcb32.regedx
                mov     esi, es:[di].pcb32.regesi
                mov     ebp, es:[di].pcb32.regebp
                mov     ds, es:[di].pcb32.regds

                push    es:[di].pcb32.regflags
                push    es:[di].pcb32.regcs
                push    es:[di].pcb32.regip
                push    es:[di].pcb32.regedi
```

```
                mov     es, es:[di].pcb32.reges
                pop     edi
                iret
cocall32        endp


; CoCall32l works just like cocall above, except the address of the pcb
; follows the call in the code stream rather than being passed in ES:DI.
; Note: this code does *not* return the caller's PCB address in ES:DI.
;

cocall32l       proc    far
                assume  ds:dseg
                push    ebp
                mov     bp, sp
                pushfd
                push    ds
                push    es
                push    edi
                push    eax
                mov     ax, dseg
                mov     ds, ax
                cli                     ;Critical region ahead.

; Save the current process' state:

                les     di, dseg:CurCoroutine
                pop     es:[di].pcb32.regeax
                mov     es:[di].pcb32.regebx, ebx
                mov     es:[di].pcb32.regecx, ecx
                mov     es:[di].pcb32.regedx, edx
                mov     es:[di].pcb32.regesi, esi
                pop     es:[di].pcb32.regedi
                pop     es:[di].pcb32.reges
                pop     es:[di].pcb32.regds
                pop     es:[di].pcb32.regflags
                pop     es:[di].pcb32.regebp
                pop     es:[di].pcb32.regip
                pop     es:[di].pcb32.regcs
                mov     es:[di].pcb32.regsp, sp
                mov     es:[di].pcb32.regss, ss

                mov     dx, es:[di].pcb32.regip ;Get return address (ptr to
                mov     cx, es:[di].pcb32.regcs ; PCB address.
                add     es:[di].pcb32.regip, 4  ;Skip ptr on return.
                mov     es, cx                  ;Get the ptr to the new pcb
                mov     di, dx                  ; address, then fetch the
                les     di, es:[di]             ; pcb val.
                mov     wp dseg:CurCoroutine, di
                mov     wp dseg:CurCoroutine+2, es

; Okay, switch to the new process:

                mov     ss, es:[di].pcb32.regss
                mov     sp, es:[di].pcb32.regsp
                mov     eax, es:[di].pcb32.regeax
                mov     ebx, es:[di].pcb32.regebx
                mov     ecx, es:[di].pcb32.regecx
                mov     edx, es:[di].pcb32.regedx
                mov     esi, es:[di].pcb32.regesi
                mov     ebp, es:[di].pcb32.regebp
                mov     ds, es:[di].pcb32.regds

                push    es:[di].pcb32.regflags
                push    es:[di].pcb32.regcs
                push    es:[di].pcb32.regip
                push    es:[di].pcb32.regedi
                mov     es, es:[di].pcb32.reges
                pop     edi
                iret


cocall32l       endp
cseg            ends
```

19.4 Multitasking

Coroutines provide a reasonable mechanism for switching between processes that must take turns. For example, the maze generation program in the previous section would generate poor mazes if the daemon processes didn't take turns removing one cell at a time from the maze. However, the coroutine paradigm isn't always suitable; not all processes need to take turns. For example, suppose you are writing an action game where the user plays against the computer. In addition, the computer player operates independently of the user in real time. This could be, for example, a space war game or a flight simulator game (where you are dog fighting other pilots). Ideally, we would like to have *two* computers. One to handle the user interaction and one for the computer player. Both systems would communicate their moves to one another during the game. If the (human) player simply sits and watches the screen, the computer player would win since it is active and the human player is not. Of course, it would considerably limit the marketability of your game were it to require two computers to play. However, you can use *multitasking* to simulate two separate computer systems on a single CPU.

The basic idea behind multitasking is that one process runs for a period of time (the *time quantum* or *time slice*) and then a timer interrupts the process. The timer ISR saves the state of the process and then switches control to another process. That process runs for its time slice and then the timer interrupt switches to another process. In this manner, each process gets some amount of computer time. Note that multitasking is very easy to implement if you have a coroutine package. All you need to do is write a timer ISR that cocalls the various processes, one per timer interrupt A timer interrupt that switches between processes is a *dispatcher*.

One decision you will need to make when designing a dispatcher is a policy for the process selection algorithm. A simple policy is to place all processes in a queue and then rotate among them. This is known as the *round-robin policy*. Since this is the policy the UCR Standard Library process package uses, we will adopt it as well. However, there are other process selection criteria, generally involving the priority of a process, available as well. See a good text on operating systems for details.

The choice of the time quantum can have a big impact on performance. Generally, you would like the time quantum to be small. The time sharing (switching between processes based on the clock) will be much smoother if you use small time quanta. For example, suppose you choose five second time quanta and you were running four processes concurrently. Each process would get five seconds; it would run very fast during those five seconds. However, at the end of its time slice it would have to wait for the other three process' turns, 15 seconds, before it ran again. The users of such programs would get very frustrated with them, users like programs whose performance is relatively consistent from one moment to the next.

If we make the time slice one millisecond, instead of five seconds, each process would run for one millisecond and then switch to the next processes. This means that each processes gets one millisecond out of five. This is too small a time quantum for the user to notice the pause between processes.

Since smaller time quanta seem to be better, you might wonder "why not make them as small as possible?" For example, the PC supports a one millisecond timer interrupt. Why not use that to switch between processes? The problem is that there is a fair amount of overhead required to switch from one processes to another. The smaller you make the time quantum, the larger will be the overhead of using time slicing. Therefore, you want to pick a time quantum that is a good balance between smooth process switching and too much overhead. As it turns out, the $^{1}/_{18}$th second clock is probably fine for most multitasking requirements.

19.4.1 Lightweight and HeavyWeight Processes

There are two major types of processes in the world of multitasking: *lightweight processes*, also known as *threads*, and *heavyweight processes*. These two types of processes differ mainly in the details of memory management. A heavyweight process swaps memory management tables and moves lots of data

around. Threads only swap the stack and CPU registers. Threads have much less overhead cost than heavyweight processes.

We will not consider heavyweight processes in this text. Heavyweight processes appear in protected mode operating systems like UNIX, Linux, OS/2, or Windows NT. Since there is rarely any memory management (at the hardware level) going on under DOS, the issue of changing memory management tables around is moot. Switching from one heavyweight application to another generally corresponds to switching from one application to another.

Using lightweight processes (threads) is perfectly reasonable under DOS. Threads (short for "execution thread" or "thread of execution") correspond to two or more concurrent execution paths within the same program. For example, we could think of each of the demons in the maze generation program as being a separate thread of execution.

Although threads have different stacks and machine states, they share code and data memory. There is no need to use a "shared memory TSR" to provide global shared memory (see "Shared Memory" on page 1078). Instead, maintaining *local* variables is the difficult task. You must either allocate local variables on the process' stack (which is separate for each process) or you've got to make sure that no other process uses the variables you declare in the data segment specifically for one thread.

We could easily write our own threads package, but we don't have to; the UCR Standard Library provides this capability in the *processes package*. To see how to incorporate threads into your programs, keep reading…

19.4.2 The UCR Standard Library Processes Package

The UCR Standard Library provides six routines to let you manage threads. These routines include `prcsinit`, `prcsquit`, `fork`, `die`, `kill`, and `yield`. These functions let you initialize and shut down the threads system, start new processes, terminate processes, and voluntarily pass the CPU off to another process.

The `prcsinit` and `prcsquit` functions let you initialize and shutdown the system. The `prcsinit` call prepares the threads package. You must call this routine before executing any of the other five process routines. The `prcsquit` function shuts down the threads system in preparation for program termination. `Prcsinit` patches into the timer interrupt (interrupt 8). `Prcsquit` restores the interrupt 8 vector. It is very important that you call `prcsquit` before your program returns to DOS. Failure to do so will leave the int 8 vector pointing off into memory which may cause the system to crash when DOS loads the next program. Your program must patch the break and critical error exception vectors to ensure that you call `prcsquit` in the event of abnormal program termination. Failure to do so may crash the system if the user terminates the program with ctrl-break or an abort on an I/O error. `Prcsinit` and `prcsquit` do not require any parameters, nor do they return any values.

The `fork` call spawns a new process. On entry, `es:di` must point at a `pcb` for the new process. The `regss` and `regsp` fields of the `pcb` must contain the address of the *top* of the stack area for this new process. The `fork` call fills in the other fields of the `pcb` (including cs:ip)/

For each call you make to `fork`, the `fork` routine returns *twice*, once for each thread of execution. The parent process *typically* returns first, but this is not certain; the child process is usually the second return from the `fork` call. To differentiate the two calls, `fork` returns two process identifiers (PIDs) in the `ax` and `bx` registers. For the parent process, `fork` returns with `ax` containing zero and `bx` containing the PID of the child process. For the child process, `fork` returns with `ax` containing the child's PID and `bx` containing zero. Note that both threads return and continuing executing the same code after the call to `fork`. If you want the child and parent processes to take separate paths, you would execute code like the following:

```
        lesi    NewPCB          ;Assume regss/regsp are initialized.
        fork
        test    ax, ax          ;Parent PID is zero at this point.
        je      ParentProcess       ;Go elsewhere if parent process.

; Child process continues execution here
```

The parent process should save the child's PID. You can use the PID to terminate a process at some later time.

It is important to repeat that you must initialize the regss and regsp fields in the pcb before calling fork. You must allocate storage for a stack (dynamically or statically) and point ss:sp at the last word of this stack area. Once you call fork, the process package uses whatever value that happens to be in the regss and regsp fields. If you have not initialized these values, they will probably contain zero and when the process starts it will wipe out the data at address 0:FFFE. This may crash the system at one point or another.

The die call kills the current process. If there are multiple processes running, this call transfers control to some other processes waiting to run. If the current process is the only process on the system's *run queue*, then this call will crash the system.

The kill call lets one process terminate another. Typically, a parent process will use this call to terminate a child process. To kill a process, simply load the ax register with the PID of the process you want to terminate and then call kill. If a process supplies its own PID to the kill function, the process terminates itself (that is, this is equivalent to a die call). If there is only one process in the run queue and that process kills itself, the system will crash.

The last multitasking management routine in the process package is the yield call. Yield voluntarily gives up the CPU. This is a direct call to the dispatcher, that will switch to another task in the run queue. Control returns after the yield call when the next time slice is given to this process. If the current process is the only one in the queue, yield immediately returns. You would normally use the yield call to free up the CPU between long I/O operations (like waiting for a keypress). This would allow other tasks to get maximum use of the CPU while your process is just spinning in a loop waiting for some I/O operation to complete.

The Standard Library multitasking routines only work with the 16 bit register set of the 80x86 family. Like the coroutine package, you will need to modify the pcb and the dispatcher code if you want to support the 32 bit register set of the 80386 and later processors. This task is relatively simple and the code is quite similar to that appearing in the section on coroutines; so there is no need to present the solution here.

19.4.3 Problems with Multitasking

When threads share code and data certain problems can develop. First of all, reentrancy becomes a problem. You cannot call a non-reentrant routine (like DOS) from two separate threads if there is ever the possibility that the non-reentrant code could be interrupted and control transferred to a second thread that reenters the same routine. Reentrancy is not the only problem, however. It is quite possible to design two routines that access shared variables and those routines misbehave depending on where the interrupts occur in the code sequence. We will explore these problems in the section on synchronization (see "Synchronization" on page 1129), just be aware, for now, that these problems exist.

Note that simply turning off the interrupts (with cli) may not solve the reentrancy problem. Consider the following code:

```
        cli                     ;Prevent reentrancy.
        mov     ah, 3Eh         ;DOS close call.
        mov     bx, Handle
        int     21h
        sti                     ;Turn interrupts back on.
```

This code will not prevent DOS from being reentered because DOS (and BIOS) turn the interrupts back on! There is a solution to this problem, but it's not by using `cli` and `sti`.

19.4.4 A Sample Program with Threads

The following program provides a simple demonstration of the Standard Library processes package. This short program creates two threads – the main program and a timer process. On each timer tick the background (timer) process kicks in and increments a memory variable. It then yields the CPU back to the main program. On the next timer tick control returns to the background process and this cycle repeats. The main program reads a string from the user while the background process is counting off timer ticks. When the user finishes the line by pressing the enter key, the main program kills the background process and then prints the amount of time necessary to enter the line of text.

Of course, this isn't the most efficient way to time how long it takes someone to enter a line of text, but it does provide an example of the multitasking features of the Standard Library. This short program segment demonstrates all the process routines except `die`. Note that it also demonstrates the fact that you must supply int 23h and int 24h handlers when using the process package.

```
; MULTI.ASM
; Simple program to demonstrate the use of multitasking.


                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment    para public 'data'

ChildPID        word       0             ;Child's PID so we can kill it.
BackGndCnt      word       0             ;Counts off clock ticks in backgnd.

; PCB for our background process. Note we initialize ss:sp here.

BkgndPCB        pcb        {0,offset EndStk2, seg EndStk2}

; Data buffer to hold an input string.

InputLine       byte       128 dup (0)

dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg

; A replacement critical error handler. This routine calls prcsquit
; if the user decides to abort the program.


CritErrMsg      byte       cr,lf
                byte       "DOS Critical Error!",cr,lf
                byte       "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24         proc       far
                push       dx
                push       ds
                push       ax

                push       cs
                pop        ds
Int24Lp:        lea        dx, CritErrMsg
                mov        ah, 9             ;DOS print string call.
                int        21h

                mov        ah, 1             ;DOS read character call.
                int        21h
```

```
                and     al, 5Fh         ;Convert l.c. -> u.c.

                cmp     al, 'I'         ;Ignore?
                jne     NotIgnore
                pop     ax
                mov     al, 0
                jmp     Quit24

NotIgnore:      cmp     al, 'r'         ;Retry?
                jne     NotRetry
                pop     ax
                mov     al, 1
                jmp     Quit24

NotRetry:       cmp     al, 'A'         ;Abort?
                jne     NotAbort
                prcsquit                ;If quitting, fix INT 8.
                pop     ax
                mov     al, 2
                jmp     Quit24

NotAbort:       cmp     al, 'F'
                jne     BadChar
                pop     ax
                mov     al, 3
Quit24:         pop     ds
                pop     dx
                iret

BadChar:        mov     ah, 2
                mov     dl, 7           ;Bell character
                jmp     Int24Lp
MyInt24         endp


; We will simply disable INT 23h (the break exception).

MyInt23         proc    far
                iret
MyInt23         endp


; Okay, this is a pretty weak background process, but it does demonstrate
; how to use the Standard Library calls.

BackGround      proc
                sti
                mov     ax, dseg
                mov     ds, ax
                inc     BackGndCnt      ;Bump call Counter by one.
                yield                   ;Give CPU back to foregnd.
                jmp     BackGround
BackGround      endp


Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit


; Initialize the INT 23h and INT 24h exception handler vectors.

                mov     ax, 0
                mov     es, ax
                mov     word ptr es:[24h*4], offset MyInt24
                mov     es:[24h*4 + 2], cs
                mov     word ptr es:[23h*4], offset MyInt23
                mov     es:[23h*4 + 2], cs

                prcsinit                ;Start multitasking system.
```

```
                lesi    BkgndPCB        ;Fire up a new process
                fork
                test    ax, ax          ;Parent's return?
                je      ParentPrcs
                jmp     BackGround      ;Go do backgroun stuff.

ParentPrcs:     mov     ChildPID, bx    ;Save child process ID.

                print
                byte    "I am timing you while you enter a string. So type"
                byte    cr,lf
                byte    "quickly: ",0

                lesi    InputLine
                gets

                mov     ax, ChildPID    ;Stop the child from running.
                kill

                printf
                byte    "While entering '%s' you took %d clock ticks"
                byte    cr,lf,0
                dword   InputLine, BackGndCnt

                prcsquit

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment stack 'stack'

; Here is the stack for the background process we start

stk2            byte    256 dup (?)
EndStk2         word    ?

;Here's the stack for the main program/foreground process.

stk             byte    1024 dup (?)
sseg            ends

zzzzzzseg       segment public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

19.5 Synchronization

Many problems occur in cooperative concurrently executing processes due to *synchronization* (or the lack thereof). For example, one process can *produce* data that other processes *consume*. However, it might take much longer for the producer to create than data than it takes for the consumer to use it. Some mechanism must be in place to ensure that the consumer does not attempt to use the data before the producer creates it. Likewise, we need to ensure that the consumer uses the data created by the producer before the producer creates more data.

The *producer-consumer problem* is one of several very famous *synchronization* problems from operating systems theory. In the producer-consumer problem there are one or more processes that produce data and write this data to a shared buffer. Likewise, there are one or more consumers that read data from this buffer. There are two synchronization issues we must deal with – the first is to ensure that the producers do not produce more data than the buffer can hold (conversely, we must prevent the consumers from removing data from an empty buffer); the second is to ensure the integrity of the buffer data structure by allowing access to only one process at a time.

Consider what can happen in a simple producer-consumer problem. Suppose the producer and consumer processes share a single data buffer structure organized as follows:

```
buffer          struct
Count           word        0
InPtr           word        0
OutPtr          word        0
Data            byte        MaxBufSize dup (?)
buffer          ends
```

The Count field specifies the number of data bytes currently in the buffer. InPtr points at the next available location to place data in the buffer. OutPtr is the address of the next byte to remove from the buffer. Data is the actual buffer array. Adding and removing data is very easy. The following code segments *almost* handle this job:

```
; Producer-   This procedure adds the value in al to the buffer.
;             Assume that the buffer variable MyBuffer is in the data segment.

Producer        proc        near
                pushf
                sti                             ;Must have interrupts on!
                push        bx

; The following loop waits until there is room in the buffer to insert
; another byte.

WaitForRoom:    cmp         MyBuffer.Count, MaxBufSize
                jae         WaitForRoom

; Okay, insert the byte into the buffer.

                mov         bx, MyBuffer.InPtr
                mov         MyBuffer.Data[bx], al
                inc         MyBuffer.Count      ;We just added a byte to the buffer.
                inc         MyBuffer.InPtr      ;Move on to next item in buffer.

; If we are at the physical end of the buffer, wrap around to the beginning.

                cmp         MyBuffer.InPtr, MaxBufSize
                jb          NoWrap
                mov         MyBuffer.InPtr, 0
NoWrap:
                pop         bx
                popf
                ret
Producer        endp


; Consumer-   This procedure waits for data (if necessary) and returns the
;             next available byte from the buffer.

Consumer        proc        near
                pushf                           ;Must have interrupts on!
                sti
                push        bx
WaitForData:    cmp         Count, 0            ;Is the buffer empty?
                je          WaitForData         ;If so, wait for data to arrive.

; Okay, fetch an input character

                mov         bx, MyBuffer.OutPtr
                mov         al, MyBuffer.Data[bx]
                dec         MyBuffer.Count
                inc         MyBuffer.OutPtr
                cmp         MyBuffer.OutPtr, MaxBufSize
                jb          NoWrap
                mov         MyBuffer.OutPtr, 0
NoWrap:
                pop         bx
                popf
                ret
Consumer        endp
```

The only problem with this code is that it won't always work if there are multiple producer or consumer processes. In fact, it is easy to come up with a version of this code that won't work for a single set of producer and consumer processes (although the code above will work fine, in that special case). The problem is that these procedures access global variables and, therefore, are not reentrant. In particular, the problem lies with the way these two procedures manipulate the buffer control variables. Consider, for a moment, the following statements from the Consumer procedure:

```
                dec         MyBuffer.Count

          « Suppose an interrupt occurs here »

                inc         MyBuffer.OutPtr
                cmp         MyBuffer.OutPtr, MaxBufSize
                jb          NoWrap
                mov         MyBuffer.OutPtr, 0
NoWrap:
```

If an interrupt occurs at the specified point above and control transfers to another consumer process that reenters this code, the second consumer would malfunction. The problem is that the first consumer has fetched data from the buffer but has yet to update the output pointer. The second consumer comes along and removes *the same byte as the first consumer*. The second consumer then properly updates the output pointer to point at the next available location in the circular buffer. When control eventually returns to the first consumer process, it finishes the operation by incrementing the output pointer. *This causes the system to skip over the next byte which no process has read.* The end result is that two consumer processes fetch the same byte and then skip a byte in the buffer.

This problem is easily solved by recognizing the fact that the code that manipulates the buffer data is a critical region. By restricting execution in the critical region to one process at a time, we can solve this problem. In the simple example above, we can easily prevent reentrancy by turning the interrupts off while in the critical region. For the consumer procedure, the code would look like this:

```
; Consumer-   This procedure waits for data (if necessary) and returns the
;             next available byte from the buffer.

Consumer      proc        near
              pushf                           ;Must have interrupts on!
              sti
              push        bx
WaitForData:  cmp         Count, 0            ;Is the buffer empty?
              je          WaitForData         ;If so, wait for data to arrive.

; The following is a critical region, so turn the interrupts off.

              cli

; Okay, fetch an input character

              mov         bx, MyBuffer.OutPtr
              mov         al, MyBuffer.Data[bx]
              dec         MyBuffer.Count
              inc         MyBuffer.OutPtr
              cmp         MyBuffer.OutPtr, MaxBufSize
              jb          NoWrap
              mov         MyBuffer.OutPtr, 0
NoWrap:
              pop         bx
              popf                            ;Restore interrupt flag.
              ret
Consumer      endp
```

Note that we cannot turn the interrupts off during the execution of the whole procedure. Interrupts must be on while this procedure is waiting for data, otherwise the producer process will never be able to put data in the buffer for the consumer.

Simply turning the interrupts off does not always work. Some critical regions may take a considerable amount of time (seconds, minutes, or even hours) and you cannot leave the interrupts off for that amount

of time[3]. Another problem is that the critical region may call a procedure that turns the interrupts back on and you have no control over this. A good example is a procedure that calls MS-DOS. Since MS-DOS is not reentrant, MS-DOS is, by definition, a critical section; we can only allow one process at a time inside MS-DOS. However, MS-DOS reenables the interrupts, so we cannot simply turn off the interrupts before calling an MS-DOS function an expect this to prevent reentrancy.

Turning off the interrupts doesn't even work for the consumer/producer procedures given earlier. Note that interrupts *must* be on while the consumer is waiting for data to arrive in the buffer (conversely, the producers must have interrupts on while waiting for room in the buffer). It is quite possible for the code to detect the presence of data and just before the execution of the `cli` instruction, an interrupt transfers control to a second consumer process. While it is not possible for both processes to update the buffer variables concurrently, it is possible for the second consumer process to remove the *only* data value from the input buffer and then switch back to the first consumer that removes a phantom value from the buffer (and causes the `Count` variable to go negative).

One poorly thought out solution is to use a flag to control access to a critical region. A process, before entering the critical region, tests the flag to see if any other process is currently in the critical region; if not, the process sets the flag to "in use" and then enters the critical region. Upon leaving the critical region, the process sets the flag to "not in use." If a process wants to enter a critical region and the flag's value is "in use", the process must wait until the process currently in the critical section finishes and writes the "not in use" value to the flag.

The only problem with this solution is that it is nothing more than a special case of the producer/consumer problem. The instructions that update the in-use flag form their own critical section that you must protect. As a general solution, the in-use flag idea fails.

19.5.1 Atomic Operations, Test & Set, and Busy-Waiting

The problem with the in-use flag idea is that it takes several instructions to test and set the flag. A typical piece of code that tests such a flag would read its value and determine if the critical section is in use. If not, it would then write the "in-use" value to the flag to let other processes know that it is in the critical section. The problem is that an interrupt could occur after the code tests the flag but before it sets the flag to "in use." Then some other process can come along, test the flag and find that it is not in use, and enter the critical region. The system could interrupt that second process while it is still in the critical region and transfer control back to the first. Since the first process has already determined that the critical region is not in use, it sets the flag to "in use" and enters the critical region. Now we have two processes in the critical region and the system is in violation of the *mutual exclusion requirement* (only one process in a critical region at a time).

The problem with this approach is that testing and setting the in-use flag is not an uninterruptable (*atomic*) operation. If it were, then there would be no problem. Of course, it is easy to make a sequence of instructions non-interruptible by putting a `cli` instruction before them. Therefore, we can test and set a flag in an atomic operation as follows (assume in-use is zero, not in-use is one):

```
            pushf
TestLoop:   cli                         ;Turn ints off while testing and
            cmp       Flag, 0           ; setting flag.
            je        IsInUse           ;Already in use?
            mov       Flag, 0           ;If not, make it so.
IsInUse:    sti                         ;Allow ints (if in-use already).
            je        TestLoop          ;Wait until not in use.
            popf

; When we get down here, the flag was "not in-use" and we've just set it
; to "in-us." We now have exclusive access to the critical section.
```

3. In general, you should not leave the interrupts off for more than about 30 milliseconds when using the 1/18th second clock for multitasking. A general rule of thumb is that interrupts should not be off for much more than abou;50% of the time quantum.

Another solution is to use a so-called "test and set" instruction – one that both tests a specific condition and sets the flag to a desired value. In our case, we need an instruction that both tests a flag to see if it is not in-use and sets it to in-use at the same time (if the flag was already in-use, it will remain in use afterward). Although the 80x86 does not support a specific test and set instruction, it does provide several others that can achieve the same effect. These instructions include xchg, shl, shr, sar, rcl, rcr, rol, ror, btc/btr/bts (available only on the 80386 and later processors), and cmpxchg (available only on the 80486 and later processors). In a limited sense, you can also use the addition and subtraction instructions (add, sub, adc, sbb, inc, and dec) as well.

The exchange instruction provides the most generic form for the test and set operation. If you have a flag (0=in use, 1=not in use) you can test and set this flag without messing with the interrupts using the following code:

```
InUseLoop:      mov         al, 0               ;0=In Use
                xchg        al, Flag
                cmp         al, 0
                je          InUseLoop
```

The xchg instruction atomically swaps the value in al with the value in the flag variable. Although the xchg instruction doesn't actually test the value, it does place the original flag value in a location (al) that is safe from modification by another process. If the flag originally contained zero (in-use), this exchange sequence swaps a zero for the existing zero and the loop repeats. If the flag originally contained a one (not in-use) then this code swaps a zero (in-use) for the one and falls out of the in use loop.

The shift and rotate instructions also act as test and set instructions, assuming you use the proper values for the in-use flag. With in-use equal to zero and not in-use equal to one, the following code demonstrates how to use the shr instruction for the test and set operation:

```
InUseLoop:      shr         Flag, 1             ;In-use bit to carry, 0->Flag.
                jnc         InUseLoop           ;Repeat if already in use.
```

This code shifts the in-use bit (bit number zero) into the carry flag and clears the in-use flag. At the same time, it zeros the Flag variable, assuming Flag always contains zero or one. The code for the atomic test and set sequences using the other shift and rotates is very similar and appears in the exercises.

Starting with the 80386, Intel provided a set of instructions explicitly intended for test and set operations: btc (bit test and complement), bts (bit test and set), and btr (bit test and reset). These instructions copy a specific bit from the destination operand into the carry flag and then complement, set, or reset (clear) that bit. The following code demonstrates how to use the btr instruction to manipulate our in-use flag:

```
InUseLoop:      btr         Flag, 0             ;In-use flag is in bit zero.
                jnc         InUseLoop
```

The btr instruction is a little more flexible than the shr instruction because you don't have to guarantee that all the other bits in the Flag variable are zero; it tests and clears bit zero without affect any other bits in the Flag variable.

The 80486 (and later) cmpxchg instruction provides a very generic synchronization primitive. A "compare and swap" instruction turns out to be the only atomic instruction you need to implement almost *any* synchronization primitive. However, its generic structure means that it is a little too complex for simple test and set operations. You will get an opportunity to design a test and set sequence using cmpxchg in the exercises. For more details on cmpxchg, see "The CMPXCHG, and CMPXCHG8B Instructions" on page 263.

Returning to the producer/consumer problem, we can easily solve the critical region problem that exists in these routines using the test and set instruction sequence presented above. The following code does this for the Producer procedure, you would modify the Consumer procedure in a similar fashion.

```
; Producer-     This procedure adds the value in al to the buffer.
;               Assume that the buffer variable MyBuffer is in the data segment.

Producer        proc        near
```

```
                pushf
                sti                             ;Must have interrupts on!

; Okay, we are about to enter a critical region (this whole procedure),
; so test the in-use flag to see if this critical region is already in use.

InUseLoop:      shr         Flag, 1
                jnc         InUseLoop

                push        bx

; The following loop waits until there is room in the buffer to insert
; another byte.

WaitForRoom:    cmp         MyBuffer.Count, MaxBufSize
                jae         WaitForRoom

; Okay, insert the byte into the buffer.

                mov         bx, MyBuffer.InPtr
                mov         MyBuffer.Data[bx], al
                inc         MyBuffer.Count    ;We just added a byte to the buffer.
                inc         MyBuffer.InPtr    ;Move on to next item in buffer.

; If we are at the physical end of the buffer, wrap around to the beginning.

                cmp         MyBuffer.InPtr, MaxBufSize
                jb          NoWrap
                mov         MyBuffer.InPtr, 0
NoWrap:
                mov         Flag, 1           ;Set flag to not in use.
                pop         bx
                popf
                ret
Producer        endp
```

One minor problem with the test and set approach to protecting a critical region is that it uses a *busy-waiting* loop. While the critical region is not available, the process spins in a loop waiting for its turn at the critical region. If the process that is currently in the critical region remains there for a considerable length of time (say, seconds, minutes, or hours), the process(es) waiting to enter the critical region continue to waste CPU time waiting for the flag. This, in turn, wastes CPU time that could be put to better use getting the process in the critical region through it so another process can enter.

Another problem that might exist is that it is possible for one process to enter the critical region, locking other processes out, leave the critical region, do some processing, and then reenter the critical region all during the same time slice. If it turns out that the process is always in the critical region when the timer interrupt occurs, none of the other processes waiting to enter the critical region will ever do so. This is a problem known as *starvation* – processes waiting to enter the critical region never do so because some other process always beats them into it.

One solution to these two problems is to use a synchronization object known as a semaphore. Semaphores provide an efficient and general purpose mechanism for protecting critical regions. To find out about semaphores, keep reading...

19.5.2 Semaphores

A semaphore is an object with two basic methods: wait and signal (or release). To use a semaphore, you create a semaphore variable (an instance) for a particular critical region or other *resource* you want to protect. When a process wants to use a given resource, it *waits* on the semaphore. If no other process is currently using the resource, then the wait call sets the semaphore to in-use and immediately returns to the process. At that time, the process has exclusive access to the resource. If some other process is already using the resource (e.g., is in the critical region), then the semaphore *blocks* the current process by moving it off the run queue and onto the semaphore queue. When the process that currently holds the

resource *releases* it, the release operation removes the first waiting process from the semaphore queue and places it back in the run queue. At the next available time slice, that new process returns from its wait call and can enter its critical region.

Semaphores solve the two important problems with the busy-waiting loop described in the previous section. First, when a process waits and the semaphore blocks the process, that process is no longer on the run queue, so it consumes no more CPU time until the point that a release operation places it back onto the run queue. So unlike busy-waiting, the semaphore mechanism does not waste (as much) CPU time on processes that are waiting for some resource.

Semaphores can also solve the starvation problem. The wait operation, when blocking a process, can place it at the end of a FIFO semaphore queue. The release operation can fetch a new process from the front of the FIFO queue to place back on to the run queue. This policy ensures that each process entering the semaphore queue gets equal priority access to the resource[4].

Implementing semaphores is an easy task. A semaphore generally consists of an integer variable and a queue. The system initializes the integer variable with the number of processes than may share the resource at one time (this value is usually one for critical regions and other resources requiring exclusive access). The wait operation decrements this variable. If the result is greater than or equal to zero, the wait function simply returns to the caller; if the result is less than zero, the wait function saves the machine state, moves the process' `pcb` from the run queue to the semaphore's queue, and then switches the CPU to a different process (i.e., a `yield` call).

The release function is almost the converse. It increments the integer value. If the result is not one, the release function moves a `pcb` from the front of the semaphore queue to the run queue. If the integer value becomes one, there are no more processes on the semaphore queue, so the release function simply returns to the caller. Note that the release function does *not* activate the process it removes from the semaphore process queue. It simply places that process in the run queue. Control always returns to the process that made the release call (unless, of course, a timer interrupt occurs while executing the release function).

Of course, any time you manipulate the system's run queue you are in a critical region. Therefore, we seem to have a minor problem here – the whole purpose of a semaphore is to protect a critical region, yet the semaphore itself has a critical region we need to protect. This seems to involve circular reasoning. However, this problem is easily solved. Remember, the main reasons we do not turn off interrupts to protect a critical region is because that critical region may take a long time to execute or it may call other routines that turn the interrupts back on. The critical section in a semaphore is very short and does not call any other routines. Therefore, briefly turning off the interrupts while in the semaphore's critical region is perfectly reasonable.

If you are not allowed to turn off interrupts, you can always use a test and set instruction in a loop to protect a critical region. Although this introduces a busy-waiting loop, it turns out that you will never wait more than two time slices before exiting the busy-waiting loop, so you do not waste much CPU time waiting to enter the semaphore's critical region.

Although semaphores solve the two major problems with the busy waiting loop, it is very easy to get into trouble when using semaphores. For example, if a process waits on a semaphore and the semaphore grants exclusive access to the associate resource, then that process never releases the semaphore, any processes waiting on that semaphore will be suspended indefinitely. Likewise, any process that waits on the same semaphore twice without a release in-between will suspend itself, and any other processes that wait on that semaphore, indefinitely. Any process that does not release a resource it no longer needs violates the concept of a semaphore and is a logic error in the program. There are also some problems that may develop if a process waits on multiple semaphores before releasing any. We will return to that problem in the section on *deadlocks* (see “Deadlock” on page 1146).

4. This FIFO policy is but one example of a release policy. You could have some other policy based on a priority scheme. However, the FIFO policy does not promote starvation.

Although we could write our own semaphore package (and there is good reason to), the Standard Library process package provides its own wait and release calls along with a definition for a semaphore variable. The next section describes those calls.

19.5.3 The UCR Standard Library Semaphore Support

The UCR Standard Library process package provides two functions to manipulate semaphore variables: WaitSemaph and RlsSemaph. These functions wait and signal a semaphore, respectively. These routines mesh with the process management facilities, making it easy to implement synchronization using semaphores in your programs.

The process package provides the following definition for a semaphore data type:

```
semaphore     struct
SemaCnt       word      1
smaphrLst     dword     ?
endsmaphrLst  dword     ?
semaphore     ends
```

The SemaCnt field determines how many more processes can share a resource (if positive), or how many processes are currently waiting for the resource (if negative). By default, this field is initialized to the value one. This allows one process at a time to use the resource protected by the semaphore. Each time a process waits on a semaphore, it decrements this field. If the decremented result is positive or zero, the wait operation immediately returns. If the decremented result is negative, then the wait operation moves the current process' pcb from the run queue to the semaphore queue defined by the smaphrLst and endsmaphrLst fields in the structure above.

Most of the time you will use the default value of one for the SemaCnt field. There are some occasions, though, when you might want to allow more than one process access to some resource. For example, suppose you've developed a multiplayer game that communicates between different machines using the serial communications port or a network adapter card. You might have an area in the game which has room for only two players at a time. For example, players could be racing to a particular "transporter" room in an alien space ship, but there is room for only two players in the transporter room at a time. By initializing the semaphore variable to two, rather than one, the wait operation would allow two players to continue at one time rather than just one. When the third player attempts to enter the transporter room, the WaitSemaph function would block the player from entering the room until one of the other players left (perhaps by "transporting out" of the room).

To use the WaitSemaph or RlsSemaph function is very easy; just load the es:di register pair with the address of desired semaphore variable and issue the appropriate function call. RlsSemaph always returns immediately (assuming a timer interrupt doesn't occur while in RlsSemaph), the WaitSemaph call returns when the semaphore will allow access to the resource it protects. Examples of these two calls appear in the next section.

Like the Standard Library coroutine and process packages, the semaphore package only preserves the 16 bit register set of the 80x86 CPU. If you want to use the 32 bit register set of the 80386 and later processors, you will need to modify the source code for the WaitSemaph and RlsSemaph functions. The code you need to change is almost identical to the code in the coroutine and process packages, so this is nearly a trivial change. Do keep in mind, though, that you will need to change this code if you use any 32 bit facilities of the 80386 and later processors.

19.5.4 Using Semaphores to Protect Critical Regions

You can use semaphores to provide mutually exclusive access to any resource. For example, if several processes want to use the printer, you can create a semaphore that allows access to the printer by only one process at a time (a good example of a process that will be in the "critical region" for several minutes

at a time). However the most common task for a semaphore is to protect a critical region from reentry. Three common examples of code you need to protect from reentry include DOS calls, BIOS calls, and various Standard Library calls. Semaphores are ideal for controlling access to these functions.

To protect DOS from reentry by several different processes, you need only create a DOSsmaph variable and issue appropriate WaitSemaph and RlsSemaph calls around the call to DOS. The following sample code demonstrates how to do this.

```
; MULTIDOS.ASM
;
; This program demonstrates how to use semaphores to protect DOS calls.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment    para public 'data'

DOSsmaph        semaphore  {}

; Macros to wait and release the DOS semaphore:

DOSWait         macro
                push       es
                push       di
                lesi       DOSsmaph
                WaitSemaph
                pop        di
                pop        es
                endm

DOSRls          macro
                push       es
                push       di
                lesi       DOSsmaph
                RlsSemaph
                pop        di
                pop        es
                endm


; PCB for our background process:

BkgndPCB        pcb        {0,offset EndStk2, seg EndStk2}

; Data the foreground and background processes print:

StrPtrs1        dword      str1_a, str1_b, str1_c, str1_d, str1_e, str1_f
                dword      str1_g, str1_h, str1_i, str1_j, str1_k, str1_l
                dword      0

str1_a          byte       "Foreground: string 'a'",cr,lf,0
str1_b          byte       "Foreground: string 'b'",cr,lf,0
str1_c          byte       "Foreground: string 'c'",cr,lf,0
str1_d          byte       "Foreground: string 'd'",cr,lf,0
str1_e          byte       "Foreground: string 'e'",cr,lf,0
str1_f          byte       "Foreground: string 'f'",cr,lf,0
str1_g          byte       "Foreground: string 'g'",cr,lf,0
str1_h          byte       "Foreground: string 'h'",cr,lf,0
str1_i          byte       "Foreground: string 'i'",cr,lf,0
str1_j          byte       "Foreground: string 'j'",cr,lf,0
str1_k          byte       "Foreground: string 'k'",cr,lf,0
str1_l          byte       "Foreground: string 'l'",cr,lf,0

StrPtrs2        dword      str2_a, str2_b, str2_c, str2_d, str2_e, str2_f
                dword      str2_g, str2_h, str2_i
                dword      0

str2_a          byte       "Background: string 'a'",cr,lf,0
str2_b          byte       "Background: string 'b'",cr,lf,0
```

```
str2_c          byte        "Background: string 'c'",cr,lf,0
str2_d          byte        "Background: string 'd'",cr,lf,0
str2_e          byte        "Background: string 'e'",cr,lf,0
str2_f          byte        "Background: string 'f'",cr,lf,0
str2_g          byte        "Background: string 'g'",cr,lf,0
str2_h          byte        "Background: string 'h'",cr,lf,0
str2_i          byte        "Background: string 'i'",cr,lf,0

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

; A replacement critical error handler. This routine calls prcsquit
; if the user decides to abort the program.


CritErrMsg      byte        cr,lf
                byte        "DOS Critical Error!",cr,lf
                byte        "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24         proc        far
                push        dx
                push        ds
                push        ax

                push        cs
                pop         ds
Int24Lp:        lea         dx, CritErrMsg
                mov         ah, 9               ;DOS print string call.
                int         21h

                mov         ah, 1               ;DOS read character call.
                int         21h
                and         al, 5Fh             ;Convert l.c. -> u.c.

                cmp         al, 'I'             ;Ignore?
                jne         NotIgnore
                pop         ax
                mov         al, 0
                jmp         Quit24

NotIgnore:      cmp         al, 'r'             ;Retry?
                jne         NotRetry
                pop         ax
                mov         al, 1
                jmp         Quit24

NotRetry:       cmp         al, 'A'             ;Abort?
                jne         NotAbort
                prcsquit                        ;If quitting, fix INT 8.
                pop         ax
                mov         al, 2
                jmp         Quit24

NotAbort:       cmp         al, 'F'
                jne         BadChar
                pop         ax
                mov         al, 3
Quit24:         pop         ds
                pop         dx
                iret

BadChar:        mov         ah, 2
                mov         dl, 7               ;Bell character
                jmp         Int24Lp
MyInt24         endp


; We will simply disable INT 23h (the break exception).

MyInt23         proc        far
                iret
MyInt23         endp
```

```
; This background process calls DOS to print several strings to the
; screen. In the meantime, the foreground process is also printing
; strings to the screen. To prevent reentry, or at least a jumble of
; characters on the screen, this code uses semaphores to protect the
; DOS calls. Therefore, each process will print one complete line
; then release the semaphore. If the other process is waiting it will
; print its line.

BackGround    proc
              mov       ax, dseg
              mov       ds, ax
              lea       bx, StrPtrs2      ;Array of str ptrs.
PrintLoop:    cmp       word ptr [bx+2], 0 ;At end of pointers?
              je        BkGndDone
              les       di, [bx]          ;Get string to print.
              DOSWait
              puts                        ;Calls DOS to print string.
              DOSRls
              add       bx, 4             ;Point at next str ptr.
              jmp       PrintLoop

BkGndDone:    die                         ;Terminate this process
BackGround    endp


Main          proc
              mov       ax, dseg
              mov       ds, ax
              mov       es, ax
              meminit


; Initialize the INT 23h and INT 24h exception handler vectors.

              mov       ax, 0
              mov       es, ax
              mov       word ptr es:[24h*4], offset MyInt24
              mov       es:[24h*4 + 2], cs
              mov       word ptr es:[23h*4], offset MyInt23
              mov       es:[23h*4 + 2], cs

              prcsinit                    ;Start multitasking system.

              lesi      BkgndPCB          ;Fire up a new process
              fork
              test      ax, ax            ;Parent's return?
              je        ParentPrcs
              jmp       BackGround        ;Go do background stuff.

; The parent process will print a bunch of strings at the same time
; the background process is doing this. We'll use the DOS semaphore
; to protect the call to DOS that PUTS makes.

ParentPrcs:   DOSWait                     ;Force the other process
              mov       cx, 0             ; to wind up waiting in
DlyLp0:       loop      DlyLp0            ; the semaphore queue by
DlyLp1:       loop      DlyLp1            ; delay for at least one
DlyLp2:       loop      DlyLp2            ; clock tick.
              DOSRls

              lea       bx, StrPtrs1      ;Array of str ptrs.
PrintLoop:    cmp       word ptr [bx+2],0 ;At end of pointers?
              je        ForeGndDone
              les       di, [bx]          ;Get string to print.
              DOSWait
              puts                        ;Calls DOS to print string.
              DOSRls
              add       bx, 4             ;Point at next str ptr.
              jmp       PrintLoop

ForeGndDone:  prcsquit
```

```
Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment    para stack 'stack'

; Here is the stack for the background process we start

stk2            byte       1024 dup (?)
EndStk2         word       ?

;Here's the stack for the main program/foreground process.

stk             byte       1024 dup (?)
sseg            ends

zzzzzzseg       segment    para public 'zzzzzz'
LastBytes       db         16 dup (?)
zzzzzzseg       ends
                end        Main
```

This program doesn't directly call DOS, but it calls the Standard Library puts routine that does. In general, you could use a single semaphore to protect all BIOS, DOS, and Standard Library calls. However, this is not particularly efficient. For example, the Standard Library pattern matching routines make no DOS calls; therefore, waiting on the DOS semaphore to do a pattern match while some other process is making a DOS call unnecessarily delays the pattern match. There is nothing wrong with having one process do a pattern match while another is making a DOS call. Unfortunately, some Standard Library routines *do* make DOS calls (puts is a good example), so you must use the DOS semaphore around such calls.

In theory, we could use separate semaphores to protect DOS, different BIOS calls, and different Standard Library calls. However, keeping track of all those semaphores within a program is a big task. Furthermore, ensuring that a call to DOS does not also invoke an unprotected BIOS routine is a difficult task. So most programmers use a single semaphore to protect all Standard Library, DOS, and BIOS calls.

19.5.5 Using Semaphores for Barrier Synchronization

Although the primary use of a semaphores is to provide exclusive access to some resource, there are other synchronization uses for semaphores as well. In this section we'll look at the use of the Standard Library's semaphores objects to create a *barrier*.

A barrier is a point in a program where a process stops and waits for other processes to synchronize (reach their respective barriers). In many respects, a barrier is the dual to a semaphore. A semaphore prevents more than *n* processes from gaining access to some resource. A barrier does not grant access until at least *n* processes are requesting access.

Given the different nature of these two synchronization methods, you might think that it would be difficult to use the WaitSemaph and RlsSemaph routines to implement barriers. However, it turns out to be quite simple. Suppose we were to initialize the semaphore's SemaCnt field to zero rather than one. When the first process waits on this semaphore, the system will immediately block that process. Likewise, each additional process that waits on this semaphore will block and wait on the semaphore queue. This would normally be a disaster since there is no active process that will signal the semaphore so it will activate the blocked processes. However, if we modify the wait call so that it checks the SemaCnt field before actually doing the wait, the nth process can skip the wait call and reactivate the other processes. Consider the following macro:

```
barrier         macro       Wait4Cnt
                local       AllHere, AllDone
                cmp         es:[di].semaphore.SemaCnt, -(Wait4Cnt-1)
                jle         AllHere
                WaitSemaph
                cmp         es:[di].semaphore.SemaCnt, 0
                je          AllDone
AllHere:        RlsSemaph
AllDone:
                endm
```

This macro expects a single parameter that should be the number of processes (including the current process) that need to be at a barrier before any of the processes can proceed. The SemaCnt field is a negative number whose absolute value determines how many processes are currently waiting on the semaphore. If a barrier requires four processes, no process can proceed until the fourth process hits the barrier; at that time the SemaCnt field will contain minus three. The macro above computes what the value of SemaCnt should be if all processes are at the barrier. If SemaCnt matches this value, it signals the semaphore that begins a chain of operations with each blocked process releasing the next. When SemaCnt hits zero, the last blocked process does not release the semaphore since there are no other processes waiting on the queue.

It is very important to remember to initialize the SemaCnt field to zero before using semaphores for barrier synchronization in this manner. If you do not initialize SemaCnt to zero, the WaitSemaph call will probably not block any of the processes.

The following sample program provides a simple example of barrier synchronization using the Standard Library's semaphore package:

```
; BARRIER.ASM
;
; This sample program demonstrates how to use the Standard Library's
; semaphore objects to synchronize several processes at a barrier.
; This program is similar to the MULTIDOS.ASM program insofar as the
; background processes all print a set of strings. However, rather than
; using an inelegant delay loop to synchronize the foreground and background
; processes, this code uses barrier synchronization to achieve this.

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment    para public 'data'

BarrierSemaph   semaphore  {0}              ;Must init SemaCnt to zero.
DOSsmaph        semaphore  {}

; Macros to wait and release the DOS semaphore:

DOSWait         macro
                push       es
                push       di
                lesi       DOSsmaph
                WaitSemaph
                pop        di
                pop        es
                endm

DOSRls          macro
                push       es
                push       di
                lesi       DOSsmaph
                RlsSemaph
                pop        di
                pop        es
                endm


; Macro to synchronize on a barrier:
```

```
Barrier         macro       Wait4Cnt
                local       AllHere, AllDone
                cmp         es:[di].semaphore.SemaCnt, -(Wait4Cnt-1)
                jle         AllHere
                WaitSemaph
                cmp         es:[di].semaphore.SemaCnt, 0
                jge         AllDone
AllHere:        RlsSemaph
AllDone:
                endm



; PCBs for our background processes:

BkgndPCB2       pcb         {0,offset EndStk2, seg EndStk2}
BkgndPCB3       pcb         {0,offset EndStk3, seg EndStk3}

; Data the foreground and background processes print:

StrPtrs1        dword       str1_a, str1_b, str1_c, str1_d, str1_e, str1_f
                dword       str1_g, str1_h, str1_i, str1_j, str1_k, str1_l
                dword       0

str1_a          byte        "Foreground: string 'a'",cr,lf,0
str1_b          byte        "Foreground: string 'b'",cr,lf,0
str1_c          byte        "Foreground: string 'c'",cr,lf,0
str1_d          byte        "Foreground: string 'd'",cr,lf,0
str1_e          byte        "Foreground: string 'e'",cr,lf,0
str1_f          byte        "Foreground: string 'f'",cr,lf,0
str1_g          byte        "Foreground: string 'g'",cr,lf,0
str1_h          byte        "Foreground: string 'h'",cr,lf,0
str1_i          byte        "Foreground: string 'i'",cr,lf,0
str1_j          byte        "Foreground: string 'j'",cr,lf,0
str1_k          byte        "Foreground: string 'k'",cr,lf,0
str1_l          byte        "Foreground: string 'l'",cr,lf,0

StrPtrs2        dword       str2_a, str2_b, str2_c, str2_d, str2_e, str2_f
                dword       str2_g, str2_h, str2_i
                dword       0

str2_a          byte        "Background 1: string 'a'",cr,lf,0
str2_b          byte        "Background 1: string 'b'",cr,lf,0
str2_c          byte        "Background 1: string 'c'",cr,lf,0
str2_d          byte        "Background 1: string 'd'",cr,lf,0
str2_e          byte        "Background 1: string 'e'",cr,lf,0
str2_f          byte        "Background 1: string 'f'",cr,lf,0
str2_g          byte        "Background 1: string 'g'",cr,lf,0
str2_h          byte        "Background 1: string 'h'",cr,lf,0
str2_i          byte        "Background 1: string 'i'",cr,lf,0

StrPtrs3        dword       str3_a, str3_b, str3_c, str3_d, str3_e, str3_f
                dword       str3_g, str3_h, str3_i
                dword       0

str3_a          byte        "Background 2: string 'j'",cr,lf,0
str3_b          byte        "Background 2: string 'k'",cr,lf,0
str3_c          byte        "Background 2: string 'l'",cr,lf,0
str3_d          byte        "Background 2: string 'm'",cr,lf,0
str3_e          byte        "Background 2: string 'n'",cr,lf,0
str3_f          byte        "Background 2: string 'o'",cr,lf,0
str3_g          byte        "Background 2: string 'p'",cr,lf,0
str3_h          byte        "Background 2: string 'q'",cr,lf,0
str3_i          byte        "Background 2: string 'r'",cr,lf,0

dseg            ends

cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

; A replacement critical error handler. This routine calls prcsquit
; if the user decides to abort the program.
```

```
CritErrMsg      byte    cr,lf
                byte    "DOS Critical Error!",cr,lf
                byte    "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24         proc    far
                push    dx
                push    ds
                push    ax

                push    cs
                pop     ds
Int24Lp:        lea     dx, CritErrMsg
                mov     ah, 9           ;DOS print string call.
                int     21h

                mov     ah, 1           ;DOS read character call.
                int     21h
                and     al, 5Fh         ;Convert l.c. -> u.c.

                cmp     al, 'I'         ;Ignore?
                jne     NotIgnore
                pop     ax
                mov     al, 0
                jmp     Quit24

NotIgnore:      cmp     al, 'r'         ;Retry?
                jne     NotRetry
                pop     ax
                mov     al, 1
                jmp     Quit24

NotRetry:       cmp     al, 'A'         ;Abort?
                jne     NotAbort
                prcsquit                ;If quitting, fix INT 8.
                pop     ax
                mov     al, 2
                jmp     Quit24

NotAbort:       cmp     al, 'F'
                jne     BadChar
                pop     ax
                mov     al, 3
Quit24:         pop     ds
                pop     dx
                iret

BadChar:        mov     ah, 2
                mov     dl, 7           ;Bell character
                jmp     Int24Lp
MyInt24         endp


; We will simply disable INT 23h (the break exception).

MyInt23         proc    far
                iret
MyInt23         endp


; This background processes call DOS to print several strings to the
; screen. In the meantime, the foreground process is also printing
; strings to the screen. To prevent reentry, or at least a jumble of
; characters on the screen, this code uses semaphores to protect the
; DOS calls. Therefore, each process will print one complete line
; then release the semaphore. If the other process is waiting it will
; print its line.

BackGround1     proc
                mov     ax, dseg
                mov     ds, ax
```

```
; Wait for everyone else to get ready:

                lesi        BarrierSemaph
                barrier     3

; Okay, start printing the strings:

                lea         bx, StrPtrs2        ;Array of str ptrs.
PrintLoop:      cmp         word ptr [bx+2],0 ;At end of pointers?
                je          BkGndDone
                les         di, [bx]            ;Get string to print.
                DOSWait
                puts                            ;Calls DOS to print string.
                DOSRls
                add         bx, 4               ;Point at next str ptr.
                jmp         PrintLoop

BkGndDone:      die
BackGround1     endp


BackGround2     proc
                mov         ax, dseg
                mov         ds, ax

                lesi        BarrierSemaph
                barrier     3

                lea         bx, StrPtrs3        ;Array of str ptrs.
PrintLoop:      cmp         word ptr [bx+2],0 ;At end of pointers?
                je          BkGndDone
                les         di, [bx]            ;Get string to print.
                DOSWait
                puts                            ;Calls DOS to print string.
                DOSRls
                add         bx, 4               ;Point at next str ptr.
                jmp         PrintLoop

BkGndDone:      die
BackGround2     endp



Main            proc
                mov         ax, dseg
                mov         ds, ax
                mov         es, ax
                meminit


; Initialize the INT 23h and INT 24h exception handler vectors.

                mov         ax, 0
                mov         es, ax
                mov         word ptr es:[24h*4], offset MyInt24
                mov         es:[24h*4 + 2], cs
                mov         word ptr es:[23h*4], offset MyInt23
                mov         es:[23h*4 + 2], cs

                prcsinit                        ;Start multitasking system.

; Start the first background process:

                lesi        BkgndPCB2           ;Fire up a new process
                fork
                test        ax, ax              ;Parent's return?
                je          StartBG2
                jmp         BackGround1         ;Go do backgroun stuff.

; Start the second background process:

StartBG2:       lesi        BkgndPCB3           ;Fire up a new process
                fork
```

```
                test        ax, ax              ;Parent's return?
                je          ParentPrcs
                jmp         BackGround2         ;Go do backgroun stuff.

; The parent process will print a bunch of strings at the same time
; the background process is doing this. We'll use the DOS semaphore
; to protect the call to DOS that PUTS makes.

ParentPrcs:     lesi        BarrierSemaph
                barrier     3

                lea         bx, StrPtrs1        ;Array of str ptrs.
PrintLoop:      cmp         word ptr [bx+2],0 ;At end of pointers?
                je          ForeGndDone
                les         di, [bx]            ;Get string to print.
                DOSWait
                puts                            ;Calls DOS to print string.
                DOSRls
                add         bx, 4               ;Point at next str ptr.
                jmp         PrintLoop

ForeGndDone:    prcsquit

Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment     para stack 'stack'

; Here are the stacks for the background processes we start

stk2            byte        1024 dup (?)
EndStk2         word        ?

stk3            byte        1024 dup (?)
EndStk3         word        ?

;Here's the stack for the main program/foreground process.

stk             byte        1024 dup (?)
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

Sample Output:

```
Background 1: string 'a'
Background 1: string 'b'
Background 1: string 'c'
Background 1: string 'd'
Background 1: string 'e'
Background 1: string 'f'
Foreground: string 'a'
Background 1: string 'g'
Background 2: string 'j'
Foreground: string 'b'
Background 1: string 'h'
Background 2: string 'k'
Foreground: string 'c'
Background 1: string 'i'
Background 2: string 'l'
Foreground: string 'd'
Background 2: string 'm'
Foreground: string 'e'
Background 2: string 'n'
Foreground: string 'f'
Background 2: string 'o'
Foreground: string 'g'
```

```
Background 2: string ‘p’
Foreground: string ‘h’
Background 2: string ‘q’
Foreground: string ‘i’
Background 2: string ‘r’
Foreground: string ‘j’
Foreground: string ‘k’
Foreground: string ‘l’
```

Note how background process number one ran for one clock period before the other processes waited on the DOS semaphore. After this initial burst, the processes all took turns calling DOS.

19.6 Deadlock

Although semaphores can solve any synchronization problems, don’t get the impression that semaphores don’t introduce problems of their own. As you’ve already seen, the improper use of semaphores can result in the indefinite suspension of processes waiting on the semaphore queue. However, even if you correctly wait and signal individual semaphores, it is quite possible for correct operations on *combinations* of semaphores to produce this same effect. Indefinite suspension of a process because of semaphore problems is a serious issue. This degenerate situation is known as *deadlock* or *deadly embrace*.

Deadlock occurs when one process holds one resource and is waiting for another while a second process is holding that other resource and waiting for the first. To see how deadlock can occur, consider the following code:

```
; Process one:

              lesi       Semaph1
              WaitSemaph

        « Assume interrupt occurs here »

              lesi       Semaph2
              WaitSemaph
               .
               .
               .

; Process two:

              lesi       Semaph2
              WaitSemaph
              lesi       Semaph1
              WaitSemaph
               .
               .
               .
```

Process one grabs the semaphore associated with Semaph1. Then a timer interrupt comes along which causes a *context switch* to process two. Process two grabs the semaphore associated with Semaph2 and then tries to get Semaph1. However, process one is already holding Semaph1, so process two blocks and waits for process one to release this semaphore. This returns control (eventually) to process one. Process one then tries to graph Semaph2. Unfortunately, process two is already holding Semaph2, so process one blocks waiting for Semaph2. Now both processes are blocked waiting for the other. Since neither process can run, neither process can release the semaphore the other needs. Both processes are deadlocked.

One easy way to prevent deadlock from occurring is to never allow a process to hold more than one semaphore at a time. Unfortunately, this is not a practical solution; many processes may need to have exclusive access to several resources at one time. However, we can devise another solution by observing the pattern that resulted in deadlock in the previous example. Deadlock came about because the two processes grabbed different semaphores and then tried to grab the semaphore that the other was holding. In

other words, they grabbed the two semaphores in a different order (process one grabbed Semaph1 first and Semaph2 second, process two grabbed Semaph2 first and Semaph1 second). It turns out that two process will never deadlock if they wait on common semaphores *in the same order*. We could modify the previous example to eliminate the possibility of deadlock thusly:

```
; Process one:

                lesi       Semaph1
                WaitSemaph
                lesi       Semaph2
                WaitSemaph
                  .
                  .
                  .

; Process two:

                lesi       Semaph1
                WaitSemaph
                lesi       Semaph2
                WaitSemaph
                  .
                  .
                  .
```

Now it doesn't matter where the interrupt occurs above, deadlock cannot occur. If the interrupt occurs between the two WaitSemaph calls in process one (as before), when process two attempts to wait on Semaph1, it will block and process one will continue with Semaph2 available.

An easy way to keep out of trouble with deadlock is to number *all* your semaphore variables and make sure that all processes acquire (wait on) semaphores from the smallest numbered semaphore to the highest. This ensures that all processes acquire the semaphores in the same order, and that ensures that deadlock cannot occurs.

Note that this policy of acquiring semaphores only applies to semaphores that a process holds concurrently. If a process needs semaphore six for a while, and then it needs semaphore two after it has released semaphore six, there is no problem acquiring semaphore two after releasing semaphore six. However, if at any point the process needs to hold *both* semaphores, it must acquire semaphore two first.

Processes may release the semaphores in any order. The order that a process releases semaphores does not affect whether deadlock can occur. Of course, processes should always release a semaphore as soon as the process is done with the resource guarded by that semaphore; there may be other processes waiting on that semaphore.

While the above scheme works and is easy to implement, it is by no means the only way to handle deadlock, nor is it always the most efficient. However, it is simple to implement and it always works. For more information on deadlocks, see a good operating systems text.

19.7 Summary

Despite the fact that DOS is not reentrant and doesn't directly support multitasking, that doesn't mean your applications can't multitask; it's just difficult to get different applications to run independently of one another under DOS.

Although DOS doesn't switch among different programs in memory, DOS certainly allows you to load multiple programs into memory at one time. The only catch is that only one such program actually executes. DOS provides several calls to load and execute ".EXE" and ".COM" files from the disk. These processes effectively behave like subroutine calls, with control returning to the program invoking such a program only after that "child" program terminates. For more details, see

- "DOS Processes" on page 1065
- "Child Processes in DOS" on page 1065

- "Load and Execute" on page 1066
- "Load Program" on page 1068
- "Loading Overlays" on page 1069
- "Terminating a Process" on page 1069
- "Obtaining the Child Process Return Code" on page 1070

Certain errors can occur during the execution of a DOS process that transfer control to exception handlers. Besides the 80x86 exceptions, DOS' *break handler* and *critical error handler* are the primary examples. Any program that patches the interrupt vectors should provide its own exception handlers for these conditions so it can restore interrupts on a ctrl-C or I/O error exception. Furthermore, well-written program always provide replacement exception handlers for these two conditions that provide better support that the default DOS handlers. For more information on DOS exceptions, see

- "Exception Handling in DOS: The Break Handler" on page 1070
- "Exception Handling in DOS: The Critical Error Handler" on page 1071
- "Exception Handling in DOS: Traps" on page 1075

When a parent process invokes a child process with the LOAD or LOADEXEC calls, the child process inherits all open files from the parent process. In particular, the child process inherits the *standard input, standard output, standard error, auxiliary I/O,* and *printer* devices. The parent process can easily redirect I/O to/from these devices before passing control to a child process. This, in effect, *redirects* the I/O during the execution of the child process. For more details, see

- "Redirection of I/O for Child Processes" on page 1075

When two DOS programs want to communicate with each other, they typically read and write data to a file. However, creating, opening, reading, and writing files is a lot of work, especially just to share a few variable values. A better alternative is to use *shared memory*. Unfortunately, DOS does not provide support to allow two programs to share a common block of memory. However, it is very easy to write a TSR that manages shared memory for various programs. For details and the complete code to two shared memory managers, see:

- "Shared Memory" on page 1078
- "Static Shared Memory" on page 1078
- "Dynamic Shared Memory" on page 1088

A coroutine call is the basic mechanism for switching control between two processes. A "cocall" operation is the equivalent of a subroutine call and return all rolled into one operation. A cocall transfers control to some other process. When some other process returns control to a coroutine (via cocall), control resumes with the first instruction after the cocall code. The UCR Standard Library provides complete coroutine support so you can easily put coroutines into your assembly language programs. For all the details on coroutines, plus a neat maze generator program that uses coroutines, see

- "Coroutines" on page 1103

Although you can use coroutines to simulate multitasking ("cooperative multitasking"), the major problem with coroutines is that each application must decide when to switch to another process via a cocall. Although this eliminates certain reentrancy and synchronization problems, deciding when and where to make such calls increases the work necessary to write multitasking applications. A better approach is to use *preemptive multitasking* where the timer interrupt performs the context switches. Reentrancy and synchronization problems develop in such a system, but with care those problems are easily overcome. For the details on true preemptive multitasking, and to see how the UCR Standard Library supports multitasking, see

- "Multitasking" on page 1124
- "Lightweight and HeavyWeight Processes" on page 1124
- "The UCR Standard Library Processes Package" on page 1125
- "Problems with Multitasking" on page 1126
- "A Sample Program with Threads" on page 1127

Preemptive multitasking opens up a Pandora's box. Although multitasking makes certain programs easier to implement, the problems of process synchronization and reentrancy rears its ugly head in a multitasking system. Many processes require some sort of synchronized access to global variables. Further, most processes will need to call DOS, BIOS, or some other routine (e.g., the Standard Library) that is not reentrant. Somehow we need to control access to such code so that multiple processes do not adversely affect one another. Synchronization is achievable using several different techniques. In some simple cases we can simply turn off the interrupts, eliminating the reentrancy problems. In other cases we can use test and set or semaphores to protect a *critical region*. For more details on these synchronization operations, see

- "Synchronization" on page 1129
- "Atomic Operations, Test & Set, and Busy-Waiting" on page 1132
- "Semaphores" on page 1134
- "The UCR Standard Library Semaphore Support" on page 1136
- "Using Semaphores to Protect Critical Regions" on page 1136
- "Using Semaphores for Barrier Synchronization" on page 1140

The use of synchronization objects, like semaphores, can introduce new problems into a system. *Deadlock* is a perfect example. Deadlock occurs when one process is holding some resource and wants another and a second process is hold the desired resource and wants the resource held by the first process[5]. You can easily avoid deadlock by controlling the order that the various processes acquire groups of semaphores. For all the details, see

- "Deadlock" on page 1146

5. Or any chain of processes where everyone in the chain is holding something that another process in the chain wants.

The PC Keyboard Chapter 20

The PC's keyboard is the primary human input device on the system. Although it seems rather mundane, the keyboard is the primary input device for most software, so learning how to program the keyboard properly is very important to application developers.

IBM and countless keyboard manufacturers have produced numerous keyboards for PCs and compatibles. Most modern keyboards provide at least 101 different keys and are reasonably compatible with the IBM PC/AT 101 Key Enhanced Keyboard. Those that do provide extra keys generally program those keys to emit a sequence of other keystrokes or allow the user to program a sequence of keystrokes on the extra keys. Since the 101 key keyboard is ubiquitous, we will assume its use in this chapter.

When IBM first developed the PC, they used a very simple interface between the keyboard and the computer. When IBM introduced the PC/AT, they completely redesigned the keyboard interface. Since the introduction of the PC/AT, almost every keyboard has conformed to the PC/AT standard. Even when IBM introduced the PS/2 systems, the changes to the keyboard interface were minor and upwards compatible with the PC/AT design. Therefore, this chapter will also limit its attention to PC/AT compatible devices since so few PC/XT keyboards and systems are still in use.

There are five main components to the keyboard we will consider in this chapter – basic keyboard information, the DOS interface, the BIOS interface, the int 9 keyboard interrupt service routine, and the hardware interface to the keyboard. The last section of this chapter will discuss how to fake keyboard input into an application.

20.1 Keyboard Basics

The PC's keyboard is a computer system in its own right. Buried inside the keyboards case is an 8042 microcontroller chip that constantly scans the switches on the keyboard to see if any keys are down. This processing goes on in parallel with the normal activities of the PC, hence the keyboard never misses a keystroke because the 80x86 in the PC is busy.

A typical keystroke starts with the user pressing a key on the keyboard. This closes an electrical contact in the switch so the microcontroller and sense that you've pressed the switch. Alas, switches (being the mechanical things that they are) do not always close (make contact) so cleanly. Often, the contacts bounce off one another several times before coming to rest making a solid contact. If the microcontroller chip reads the switch constantly, these bouncing contacts will look like a very quick series of key presses and releases. This could generate *multiple* keystrokes to the main computers, a phenomenon known as *keybounce*, common to many cheap and old keyboards. But even on the most expensive and newest keyboards, keybounce is a problem if you look at the switch a million times a second; mechanical switches simply cannot settle down that quickly. Most keyboard scanning algorithms, therefore, control how often they scan the keyboard. A typical inexpensive key will settle down within five milliseconds, so if the keyboard scanning software only looks at the key every ten milliseconds, or so, the controller will effectively miss the keybounce[1].

Simply noting that a key is pressed is not sufficient reason to generate a key code. A user may hold a key down for many tens of milliseconds before releasing it. The keyboard controller must not generate a new key sequence every time it scans the keyboard and finds a key held down. Instead, it should generate a single key code value when the key goes from an up position to the down position (a *down key* operation). Upon detecting a down key stroke, the microcontroller sends a keyboard *scan code* to the PC. The scan code is *not* related to the ASCII code for that key, it is an arbitrary value IBM chose when they first developed the PC's keyboard.

1. A typical user cannot type 100 characters/sec nor reliably press a key for less than 1/50th of a second, so scanning the keyboard at 10 msec intervals will not lose any keystrokes.

The PC keyboard actually generates *two* scan codes for every key you press. It generates a *down code* when you press a key and an *up code* when you release the key. The 8042 microcontroller chip transmits these scan codes to the PC where they are processed by the keyboard's interrupt service routine. Having separate up and down codes is important because certain keys (like shift, control, and alt) are only meaningful when held down. By generating up codes for all the keys, the keyboard ensures that the keyboard interrupt service routine knows which keys are pressed while the user is holding down one of these *modifier* keys. The following table lists the scan codes that the keyboard microcontroller transmits to the PC:

Table 72: PC Keyboard Scan Codes (in hex)

Key	Down	Up	Key	Down	Up	Key	Down	Up	Key	Down	Up
Esc	1	81	[{	1A	9A	, <	33	B3	***center***	4C	CC
1 !	2	82	] }	1B	9B	. >	34	B4	***right***	4D	CD
2 @	3	83	Enter	1C	9C	/ ?	35	B5	+	4E	CE
3 #	4	84	Ctrl	1D	9D	R shift	36	B6	***end***	4F	CF
4 $	5	85	A	1E	9E	* PrtSc	37	B7	***down***	50	D0
5 %	6	86	S	1F	9F	alt	38	B8	***pgdn***	51	D1
6 ^	7	87	D	20	A0	space	39	B9	***ins***	52	D2
7 &	8	88	F	21	A1	CAPS	3A	BA	***del***	53	D3
8 *	9	89	G	22	A2	F1	3B	BB	***/***	E0 35	B5
9 (	0A	8A	H	23	A3	F2	3C	BC	***enter***	E0 1C	9C
0)	0B	8B	J	24	A4	F3	3D	BD	F11	57	D7
- _	0C	8C	K	25	A5	F4	3E	BE	F12	58	D8
= +	0D	8D	L	26	A6	F5	3F	BF	ins	E0 52	D2
Bksp	0E	8E	; :	27	A7	F6	40	C0	del	E0 53	D3
Tab	0F	8F	‘ “	28	A8	F7	41	C1	home	E0 47	C7
Q	10	90	` ~	29	A9	F8	42	C2	end	E0 4F	CF
W	11	91	L shift	2A	AA	F9	43	C3	pgup	E0 49	C9
E	12	92	\ |	2B	AB	F10	44	C4	pgdn	E0 51	D1
R	13	93	Z	2C	AC	NUM	45	C5	left	E0 4B	CB
T	14	94	X	2D	AD	SCRL	46	C6	right	E0 4D	CD
Y	15	95	C	2E	AE	***home***	47	C7	up	E0 48	C8
U	16	96	V	2F	AF	***up***	48	C8	down	E0 50	D0
I	17	97	B	30	B0	***pgup***	49	C9	R alt	E0 38	B8
O	18	98	N	31	B1	***-***	4A	CA	R ctrl	E0 1D	9D
P	19	99	M	32	B2	***left***	4B	CB	Pause	E1 1D 45 E1 9D C5	-

The keys in italics are found on the numeric keypad. Note that certain keys transmit two or more scan codes to the system. The keys that transmit more than one scan code were new keys added to the keyboard when IBM designed the 101 key enhanced keyboard.

When the scan code arrives at the PC, a second microcontroller chip receives the scan code, does a conversion on the scan code[2], makes the scan code available at I/O port 60h, and then interrupts the processor and leaves it up to the keyboard ISR to fetch the scan code from the I/O port.

The keyboard (int 9) interrupt service routine reads the scan code from the keyboard input port and processes the scan code as appropriate. Note that the scan code the system receives from the keyboard microcontroller is a single value, even though some keys on the keyboard represent up to four different values. For example, the "A" key on the keyboard can produce A, a, ctrl-A, or alt-A. The actual code the system yields depends upon the current state of the modifier keys (shift, ctrl, alt, capslock, and numlock). For example, if an A key scan code comes along (1Eh) and the shift key is down, the system produces the ASCII code for an uppercase A. If the user is pressing *multiple* modifier keys the system prioritizes them from low to high as follows:

- No modifier key down
- Numlock/Capslock (same precedence, lowest priority)
- shift
- ctrl
- alt (highest priority)

Numlock and capslock affect different sets of keys[3], so there is no ambiguity resulting from their equal precedence in the above chart. If the user is pressing two modifier keys at the same time, the system only recognizes the modifier key with the highest priority above. For example, if the user is pressing the ctrl and alt keys at the same time, the system only recognizes the alt key. The numlock, capslock, and shift keys are a special case. If numlock or capslock is active, pressing the shift key makes it inactive. Likewise, if numlock or capslock is inactive, pressing the shift key effectively "activates" these modifiers.

Not all modifiers are legal for every key. For example, ctrl-8 is not a legal combination. The keyboard interrupt service routine ignores all keypresses combined with illegal modifier keys. For some unknown reason, IBM decided to make certain key combinations legal and others illegal. For example, ctrl-left and ctrl-right are legal, but ctrl-up and ctrl-down are not. You'll see how to fix this problem a little later.

The shift, ctrl, and alt keys are *active* modifiers. That is, modification to a keypress occurs only while the user holds down one of these modifier keys. The keyboard ISR keeps track of whether these keys are down or up by setting an associated bit upon receiving the down code and clearing that bit upon receiving the up code for shift, ctrl, or alt. In contrast, the numlock, scroll lock, and capslock keys are *toggle* modifiers[4]. The keyboard ISR inverts an associated bit every time it sees a down code followed by an up code for these keys.

Most of the keys on the PC's keyboard correspond to ASCII characters. When the keyboard ISR encounters such a character, it translates it to a 16 bit value whose L.O. byte is the ASCII code and the H.O. byte is the key's scan code. For example, pressing the "A" key with no modifier, with shift, and with control produces 1E61h, 1E41h, and 1E01h, respectively ("a", "A", and ctrl-A). Many key sequences do not have corresponding ASCII codes. For example, the function keys, the cursor control keys, and the alt key sequences do not have corresponding ASCII codes. For these special *extended* code, the keyboard ISR stores a zero in the L.O. byte (where the ASCII code typically goes) and the extended code goes in the H.O. byte. The extended code is usually, though certainly not always, the scan code for that key.

The only problem with this extended code approach is that the value zero is a legal ASCII character (the NUL character). Therefore, you cannot directly enter NUL characters into an application. If an application must input NUL characters, IBM has set aside the extended code 0300h (ctrl-3) for this purpose. You application must explicitly convert this extended code to the NUL character (actually, it need only recog-

2. The keyboard doesn't actually transmit the scan codes appearing in the previous table. Instead, it transmits its own scan code that the PC's microcontroller translates to the scan codes in the table. Since the programmer never sees the native scan codes so we will ignore them.
3. Numlock only affects the keys on the numeric keypad, capslock only affects the alphabetic keys.
4. It turns out the INS key is also a toggle modifier, since it toggles a bit in the BIOS variable area. However, INS also returns a scan code, the other modifiers do not.

nize the H.O. value 03, since the L.O. byte already is the NUL character). Fortunately, very few programs need to allow the input of the NUL character from the keyboard, so this problem is rarely an issue.

The following table lists the scan and extended key codes the keyboard ISR generates for applications in response to a keypress with various modifiers. Extended codes are in italics. All other values (except the scan code column) represent the L.O. eight bits of the 16 bit code. The H.O. byte comes from the scan code column.

Table 73: Keyboard Codes (in hex)

Key	Scan Code	ASCII	Shift[a]	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num
Esc	01	1B	1B	1B		1B	1B	1B	1B
1 !	02	31	21		***7800***	31	31	31	31
2 @	03	32	40	***0300***	***7900***	32	32	32	32
3 #	04	33	23		***7A00***	33	33	33	33
4 $	05	34	24		***7B00***	34	34	34	34
5 %	06	35	25		***7C00***	35	35	35	35
6 ^	07	36	5E	1E	***7D00***	36	36	36	36
7 &	08	37	26		***7E00***	37	37	37	37
8 *	09	38	2A		***7F00***	38	38	38	38
9 (	0A	39	28		***8000***	39	39	39	39
0)	0B	30	29		***8100***	30	30	30	30
- _	0C	2D	5F	1F	***8200***	2D	2D	5F	5F
= +	0D	3D	2B		***8300***	3D	3D	2B	2B
Bksp	0E	08	08	7F		08	08	08	08
Tab	0F	09	***0F00***			09	09	***0F00***	***0F00***
Q	10	71	51	11	***1000***	71	51	71	51
W	11	77	57	17	***1100***	77	57	77	57
E	12	65	45	05	***1200***	65	45	65	45
R	13	72	52	12	***1300***	72	52	72	52
T	14	74	54	14	***1400***	74	54	74	54
Y	15	79	59	19	***1500***	79	59	79	59
U	16	75	55	15	***1600***	75	55	75	55
I	17	69	49	09	***1700***	69	49	69	49
O	18	6F	4F	0F	***1800***	6F	4F	6F	4F
P	19	70	50	10	***1900***	70	50	70	50
[{	1A	5B	7B	1B		5B	5B	7B	7B
] }	1B	5D	7D	1D		5D	5D	7D	7D
enter	1C	0D	0D	0A		0D	0D	0A	0A
ctrl	1D								
A	1E	61	41	01	***1E00***	61	41	61	41
S	1F	73	53	13	***1F00***	73	53	73	53
D	20	64	44	04	***2000***	64	44	64	44
F	21	66	46	06	***2100***	66	46	66	46
G	22	67	47	07	***2200***	67	47	67	47
H	23	68	48	08	***2300***	68	48	68	48
J	24	6A	4A	0A	***2400***	6A	4A	6A	4A
K	25	6B	4B	0B	***2500***	6B	4B	6B	4B
L	26	6C	4C	0C	***2600***	6C	4C	6C	4C
; :	27	3B	3A			3B	3B	3A	3A
‘ “	28	27	22			27	27	22	22
Key	Scan Code	ASCII	Shift	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num

Table 73: Keyboard Codes (in hex)

Key	Scan Code	ASCII	Shift[a]	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num
` ~	29	60	7E			60	60	7E	7E
Lshift	2A								
\ |	2B	5C	7C	1C		5C	5C	7C	7C
Z	2C	7A	5A	1A	***2C00***	7A	5A	7A	5A
X	2D	78	58	18	***2D00***	78	58	78	58
C	2E	63	43	03	***2E00***	63	43	63	43
V	2F	76	56	16	***2F00***	76	56	76	56
B	30	62	42	02	***3000***	62	42	62	42
N	31	6E	4E	0E	***3100***	6E	4E	6E	4E
M	32	6D	4D	0D	***3200***	6D	4D	6D	4D
, <	33	2C	3C			2C	2C	3C	3C
. >	34	2E	3E			2E	2E	3E	3E
/ ?	35	2F	3F			2F	2F	3F	3F
Rshift	36								
* PrtSc	37	2A	INT 5[b]	10[c]		2A	2A	INT 5	INT 5
alt	38								
space	39	20	20	20		20	20	20	20
caps	3A								
F1	3B	***3B00***	***5400***	***5E00***	***6800***	***3B00***	***3B00***	***5400***	***5400***
F2	3C	***3C00***	***5500***	***5F00***	***6900***	***3C00***	***3C00***	***5500***	***5500***
F3	3D	***3D00***	***5600***	***6000***	***6A00***	***3D00***	***3D00***	***5600***	***5600***
F4	3E	***3E00***	***5700***	***6100***	***6B00***	***3E00***	***3E00***	***5700***	***5700***
F5	3F	***3F00***	***5800***	***6200***	***6C00***	***3F00***	***3F00***	***5800***	***5800***
F6	40	***4000***	***5900***	***6300***	***6D00***	***4000***	***4000***	***5900***	***5900***
F7	41	***4100***	***5A00***	***6400***	***6E00***	***4100***	***4100***	***5A00***	***5A00***
F8	42	***4200***	***5B00***	***6500***	***6F00***	***4200***	***4200***	***5B00***	***5B00***
F9	43	***4300***	***5C00***	***6600***	***7000***	***4300***	***4300***	***5C00***	***5C00***
F10	44	***4400***	***5D00***	***6700***	***7100***	***4400***	***4400***	***5D00***	***5D00***
num	45								
scrl	46								
home	47	***4700***	37	***7700***		37	4700	37	4700
up	48	***4800***	38			38	4800	38	4800
pgup	49	***4900***	39	***8400***		39	4900	39	4900
-[d]	4A	2D	2D			2D	2D	2D	2D
left	4B	***4B00***	34	***7300***		34	4B00	34	4B00
center	4C	***4C00***	35			35	4C00	35	4C00
right	4D	***4D00***	36	***7400***		36	4D00	36	4D00
+[e]	4E	2B	2B			2B	2B	2B	2B
end	4F	***4F00***	31	***7500***		31	4F00	31	4F00
down	50	***5000***	32			32	5000	32	5000
pgdn	51	***5100***	33	***7600***		33	5100	33	5100
ins	52	***5200***	30			30	5200	30	5200
del	53	***5300***	2E			2E	5300	2E	5300
Key	Scan Code	ASCII	Shift	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num

a. For the alphabetic characters, if capslock is active then see the shift-capslock column.
b. Pressing the PrtSc key does not produce a scan code. Instead, BIOS executes an int 5 instruction which should print the screen.
c. This is the control-P character that will activate the printer under MS-DOS.
d. This is the minus key on the keypad.
e. This is the plus key on the keypad.

The 101-key keyboards generally provide an enter key and a "/" key on the numeric keypad. Unless you write your own int 9 keyboard ISR, you will not be able to differentiate these keys from the ones on the main keyboard. The separate cursor control pad also generates the same extended codes as the numeric keypad, except it never generates numeric ASCII codes. Otherwise, you cannot differentiate these keys from the equivalent keys on the numeric keypad (assuming numlock is off, of course).

The keyboard ISR provides a special facility that lets you enter the ASCII code for a keystroke directly from the keyboard. To do this, hold down the alt key and typing out the *decimal* ASCII code (0..255) for a character on the numeric keypad. The keyboard ISR will convert these keystrokes to an eight-bit value, attach at H.O. byte of zero to the character, and use that as the character code.

The keyboard ISR inserts the 16 bit value into the PC's *type ahead buffer*. The system type ahead buffer is a circular queue that uses the following variables

```
40:1A -  HeadPtr word ?
40:1C -  TailPtr word ?
40:1E -  Buffer  word 16 dup (?)
```

The keyboard ISR inserts data at the location pointed at by `TailPtr`. The BIOS keyboard function removes characters from the location pointed at by the `HeadPtr` variable. These two pointers almost always contain an offset into the `Buffer` array[5]. If these two pointers are equal, the type ahead buffer is empty. If the value in `HeadPtr` is two greater than the value in `TailPtr` (or `HeadPtr` is 1Eh and `TailPtr` is 3Ch), then the buffer is full and the keyboard ISR will reject any additional keystrokes.

Note that the `TailPtr` variable always points at the next available location in the type ahead buffer. Since there is no "count" variable providing the number of entries in the buffer, we must always leave one entry free in the buffer area; this means the type ahead buffer can only hold 15 keystrokes, not 16.

In addition to the type ahead buffer, the BIOS maintains several other keyboard-related variables in segment 40h. The following table lists these variables and their contents:

Table 74: Keyboard Related BIOS Variables

Name	Address[a]	Size	Description
KbdFlags1 (modifier flags)	40:17	Byte	This byte maintains the current status of the modifier keys on the keyboard. The bits have the following meanings: bit 7: Insert mode toggle bit 6: Capslock toggle (1=capslock on) bit 5: Numlock toggle (1=numlock on) bit 4: Scroll lock toggle (1=scroll lock on) bit 3: Alt key (1=alt is down) bit 2: Ctrl key (1=ctrl is down) bit 1: Left shift key (1=left shift is down) bit 0: Right shift key (1=right shift is down)

5. It is possible to change these pointers so they point elsewhere in the 40H segment, but this is not a good idea because many applications assume that these two pointers contain a value in the range 1Eh..3Ch.

Table 74: Keyboard Related BIOS Variables

Name	Address[a]	Size	Description
KbdFlags2 (Toggle keys down)	40:18	Byte	Specifies if a toggle key is currently down. bit 7: Insert key (currently down if 1) bit 6: Capslock key (currently down if 1) bit 5: Numlock key (currently down if 1) bit 4: Scroll lock key (currently down if 1) bit 3: Pause state locked (ctrl-Numlock) if one bit 2: SysReq key (currently down if 1) bit 1: Left alt key (currently down if 1) bit 0: Left ctrl key (currently down if 1)
AltKpd	40:19	Byte	BIOS uses this to compute the ASCII code for an alt--Keypad sequence.
BufStart	40:80	Word	Offset of start of keyboard buffer (1Eh). Note: this variable is not supported on many systems, be careful if you use it.
BufEnd	40:82	Word	Offset of end of keyboard buffer (3Eh). See the note above.
KbdFlags3	40:96	Byte	Miscellaneous keyboard flags. bit 7: Read of keyboard ID in progress bit 6: Last char is first kbd ID character bit 5: Force numlock on reset bit 4: 1 if 101-key kbd, 0 if 83/84 key kbd. bit 3: Right alt key pressed if 1 bit 2: Right ctrl key pressed if 1 bit 1: Last scan code was E0h bit 0: Last scan code was E1h
KbdFlags4	40:97	Byte	More miscellaneous keyboard flags. bit 7: Keyboard transmit error bit 6: Mode indicator update bit 5: Resend receive flag bit 4: Acknowledge received bit 3: Must always be zero bit 2: Capslock LED (1=on) bit 1: Numlock LED (1=on) bit 0: Scroll lock LED (1=on)

a. Addresses are all given in hexadecimal

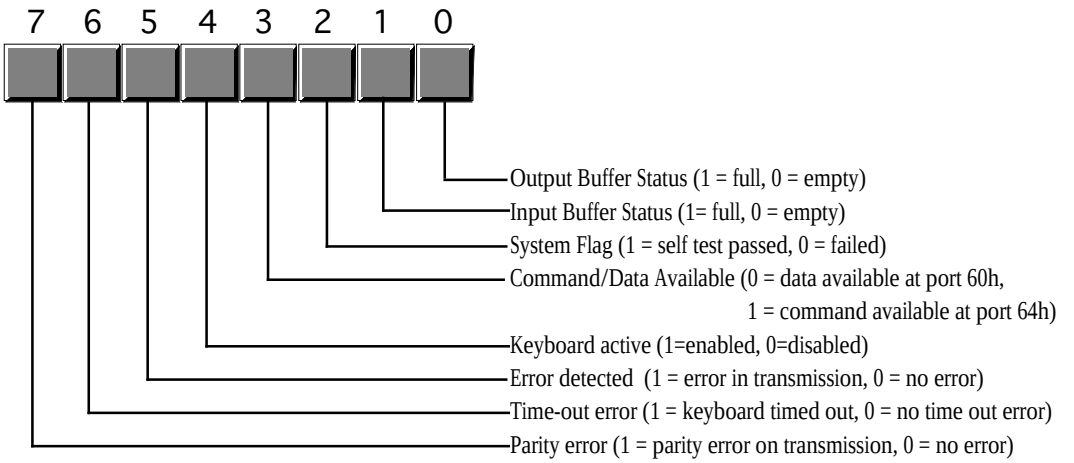
One comment is in order about KbdFlags1 and KbdFlags4. Bits zero through two of the KbdFlags4 variable is BIOS' current settings for the LEDs on the keyboard. periodically, BIOS compares the values for capslock, numlock, and scroll lock in KbdFlags1 against these three bits in KbdFlags4. If they do not agree, BIOS will send an appropriate command to the keyboard to update the LEDs and it will change the values in the KbdFlags4 variable so the system is consistent. Therefore, if you mask in new values for numlock, scroll lock, or caps lock, the BIOS will automatically adjust KbdFlags4 and set the LEDs accordingly.

20.2 The Keyboard Hardware Interface

IBM used a very simple hardware design for the keyboard port on the original PC and PC/XT machines. When they introduced the PC/AT, IBM completely resigned the interface between the PC and

the keyboard. Since then, almost every PC model and PC clone has followed this keyboard interface standard[6]. Although IBM extended the capabilities of the keyboard controller when they introduced their PS/2 systems, the PS/2 models are still upwards compatible from the PC/AT design. Since there are so few original PCs in use today (and fewer people write original software for them), we will ignore the original PC keyboard interface and concentrate on the AT and later designs.

There are two keyboard microcontrollers that the system communicates with – one on the PC's motherboard (the *on-board* microcontroller) and one inside the keyboard case (the *keyboard* microcontroller). Communication with the on-board microcontroller is through I/O port 64h. Reading this byte provides the status of the keyboard controller. Writing to this byte sends the on-board microcontroller a command. The organization of the status byte is



On-Board 8042 Keyboard Microcontroller Status Byte (Read Port 64h)

Communication to the microcontroller in the keyboard unit is via the bytes at I/O addresses 60h and 64h. Bits zero and one in the status byte at port 64h provide the necessary *handshaking* control for these ports. Before writing any data to these ports, bit zero of port 64h must be zero; data is available for reading from port 60h when bit one of port 64h contains a one. The keyboard enable and disable bits in the command byte (port 64h) determine whether the keyboard is active and whether the keyboard will interrupt the system when the user presses (or releases) a key, etc.

Bytes written to port 60h are sent to the keyboard microcontroller and bytes written to port 64h are sent to the on-board microcontroller. Bytes read from port 60h generally come from the keyboard, although you can program the on-board microcontroller to return certain values at this port, as well. The following tables lists the commands sent to the keyboard microcontroller and the values you can expect back. The following table lists the allowable commands you can write to port 64h:

Table 75: On-Board Keyboard Controller Commands (Port 64h)

Value (hex)	Description
20	Transmit keyboard controller's command byte to system as a scan code at port 60h.
60	The next byte written to port 60h will be stored in the keyboard controller's command byte.

6. We will ignore the PCjr machine in this discussion.

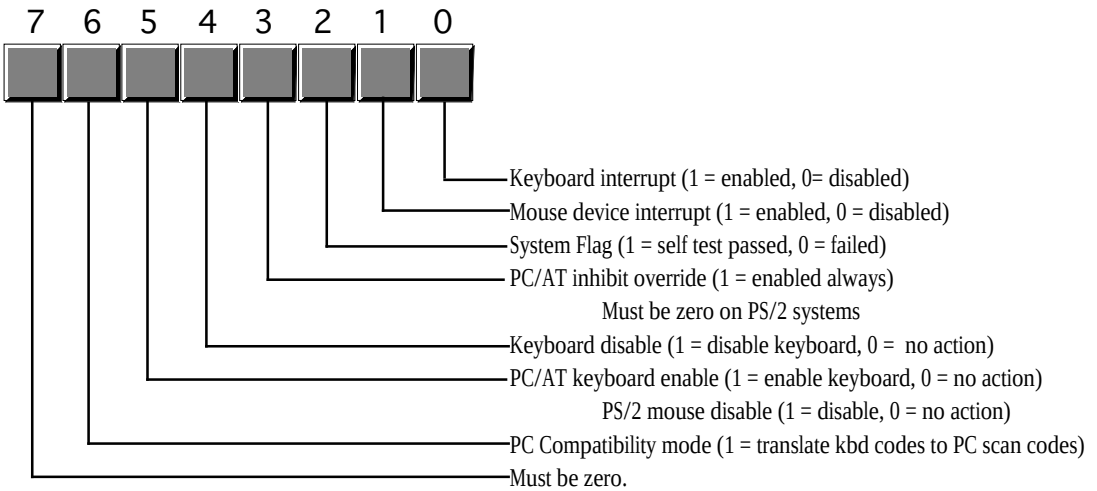
Table 75: On-Board Keyboard Controller Commands (Port 64h)

Value (hex)	Description
A4	Test if a password is installed (PS/2 only). Result comes back in port 60h. 0FAh means a password is installed, 0F1h means no password.
A5	Transmit password (PS/2 only). Starts receipt of password. The next sequence of scan codes written to port 60h, ending with a zero byte, are the new password.
A6	Password match. Characters from the keyboard are compared to password until a match occurs.
A7	Disable mouse device (PS/2 only). Identical to setting bit five of the command byte.
A8	Enable mouse device (PS/2 only). Identical to clearing bit five of the command byte.
A9	Test mouse device. Returns 0 if okay, 1 or 2 if there is a stuck clock, 3 or 4 if there is a stuck data line. Results come back in port 60h.
AA	Initiates self-test. Returns 55h in port 60h if successful.
AB	Keyboard interface test. Tests the keyboard interface. Returns 0 if okay, 1 or 2 if there is a stuck clock, 3 or 4 if there is a stuck data line. Results come back in port 60h.
AC	Diagnostic. Returns 16 bytes from the keyboard's microcontroller chip. Not available on PS/2 systems.
AD	Disable keyboard. Same operation as setting bit four of the command register.
AE	Enable keyboard. Same operation as clearing bit four of the command register.
C0	Read keyboard input port to port 60h. This input port contains the following values: bit 7: Keyboard inhibit keyswitch (0 = inhibit, 1 = enabled). bit 6: Display switch (0=color, 1=mono). bit 5: Manufacturing jumper. bit 4: System board RAM (always 1). bits 0-3: undefined.
C1	Copy input port (above) bits 0-3 to status bits 4-7. (PS/2 only)
C2	Copy input pot (above) bits 4-7 to status port bits 4-7. (PS/2 only).
D0	Copy microcontroller output port value to port 60h (see definition below).
D1	Write the next data byte written to port 60h to the microcontroller output port. This port has the following definition: bit 7: Keyboard data. bit 6: Keyboard clock. bit 5: Input buffer empty flag. bit 4: Output buffer full flag. bit 3: Undefined. bit 2: Undefined. bit 1: Gate A20 line. bit 0: System reset (if zero). Note: writing a zero to bit zero will reset the machine. Writing a one to bit one combines address lines 19 and 20 on the PC's address bus.
D2	Write keyboard buffer. The keyboard controller returns the next value sent to port 60h as though a keypress produced that value. (PS/2 only).
D3	Write mouse buffer. The keyboard controller returns the next value sent to port 60h as though a mouse operation produced that value. (PS/2 only).
D4	Writes the next data byte (60h) to the mouse (auxiliary) device. (PS/2 only).

Table 75: On-Board Keyboard Controller Commands (Port 64h)

Value (hex)	Description
E0	Read test inputs. Returns in port 60h the status of the keyboard serial lines. Bit zero contains the keyboard clock input, bit one contains the keyboard data input.
F*x*	Pulse output port (see definition for D1). Bits 0-3 of the keyboard controller command byte are pulsed onto the output port. Resets the system if bit zero is a zero.

Commands 20h and 60h let you read and write the *keyboard controller command byte*. This byte is internal to the on-board microcontroller and has the following layout:



On-Board 8042 Keyboard Microcontroller Command byte (see commands 20h and 60h)

The system transmits bytes written to I/O port 60h directly to the keyboard's microcontroller. Bit zero of the status register must contain a zero before writing any data to this port. The commands the keyboard recognizes are

Table 76: Keyboard Microcontroller Commands (Port 60h)

Value (hex)	Description
ED	Send LED bits. The next byte written to port 60h updates the LEDs on the keyboard. The parameter (next) byte contains: bits 3-7: Must be zero. bit 2: Capslock LED (1 = on, 0 = off). bit 1: Numlock LED (1 = on, 0 = off). bit 0: Scroll lock LED (1 = on, 0 = off).
EE	Echo commands. Returns 0EEh in port 60h as a diagnostic aid.

Table 76: Keyboard Microcontroller Commands (Port 60h)

Value (hex)	Description
F0	Select alternate scan code set (PS/2 only). The next byte written to port 60h selects one of the following options: 00: Report current scan code set in use (next value read from port 60h). 01: Select scan code set #1 (standard PC/AT scan code set). 02: Select scan code set #2. 03: Select scan code set #3.
F2	Send two-byte keyboard ID code as the next two bytes read from port 60h (PS/2 only).
F3	Set Autorepeat delay and repeat rate. Next byte written to port 60h determines rate: bit 7: must be zero bits 5,6: Delay. 00- $^1/_4$ sec, 01- $^1/_2$ sec, 10- $^3/_4$ sec, 11- 1 sec. bits 0-4: Repeat rate. 0- approx 30 chars/sec to 1Fh- approx 2 chars/sec.
F4	Enable keyboard.
F5	Reset to power on condition and wait for enable command.
F6	Reset to power on condition and begin scanning keyboard.
F7	Make all keys autorepeat (PS/2 only).
F8	Set all keys to generate an up code and a down code (PS/2 only).
F9	Set all keys to generate an up code only (PS/2 only).
FA	Set all keys to autorepeat and generate up and down codes (PS/2 only).
FB	Set an individual key to autorepeat. Next byte contains the scan code of the desired key. (PS/2 only).
FC	Set an individual key to generate up and down codes. Next byte contains the scan code of the desired key. (PS/2 only).
FD	Set an individual key to generate only down codes. Next byte contains the scan code of the desired key. (PS/2 only).
FE	Resend last result. Use this command if there is an error receiving data.
FF	Reset keyboard to power on state and start the self-test.

The following short program demonstrates how to send commands to the keyboard's controller. This little TSR utility programs a "light show" on the keyboard's LEDs.

```
; LEDSHOW.ASM
;
; This short TSR creates a light show on the keyboard's LEDs. For space
; reasons, this code does not implement a multiplex handler nor can you
; remove this TSR once installed. See the chapter on resident programs
; for details on how to do this.
;
; cseg and EndResident must occur before the standard library segments!

cseg            segment    para public 'code'
cseg            ends

; Marker segment, to find the end of the resident section.

EndResident     segment    para public 'Resident'
EndResident     ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list
```

```
byp             equ         <byte ptr>

cseg            segment     para public 'code'
                assume      cs:cseg, ds:cseg



; SetCmd-       Sends the command byte in the AL register to the 8042
;               keyboard microcontroller chip (command register at
;               port 64h).

SetCmd          proc        near
                push        cx
                push        ax                  ;Save command value.
                cli                             ;Critical region, no ints now.

; Wait until the 8042 is done processing the current command.

                xor         cx, cx              ;Allow 65,536 times thru loop.
Wait4Empty:     in          al, 64h             ;Read keyboard status register.
                test        al, 10b             ;Input buffer full?
                loopnz      Wait4Empty          ;If so, wait until empty.

; Okay, send the command to the 8042:

                pop         ax                  ;Retrieve command.
                out         64h, al
                sti                             ;Okay, ints can happen again.
                pop         cx
                ret
SetCmd          endp



; SendCmd-      The following routine sends a command or data byte to the
;               keyboard data port (port 60h).

SendCmd         proc        near
                push        ds
                push        bx
                push        cx
                mov         cx, 40h
                mov         ds, cx
                mov         bx, ax              ;Save data byte

                mov         al, 0ADh            ;Disable kbd for now.
                call        SetCmd

                cli                             ;Disable ints while accessing HW.

; Wait until the 8042 is done processing the current command.

                xor         cx, cx              ;Allow 65,536 times thru loop.
Wait4Empty:     in          al, 64h             ;Read keyboard status register.
                test        al, 10b             ;Input buffer full?
                loopnz      Wait4Empty          ;If so, wait until empty.

; Okay, send the data to port 60h

                mov         al, bl
                out         60h, al

                mov         al, 0AEh            ;Reenable keyboard.
                call        SetCmd
                sti                             ;Allow interrupts now.

                pop         cx
                pop         bx
                pop         ds
                ret
SendCmd         endp
```

```
; SetLEDs-    Writes the value in AL to the LEDs on the keyboard.
;             Bits 0..2 correspond to scroll, num, and caps lock,
;             respectively.

SetLEDs       proc      near
              push      ax
              push      cx

              mov       ah, al          ;Save LED bits.

              mov       al, 0EDh        ;8042 set LEDs cmd.
              call      SendCmd         ;Send the command to 8042.
              mov       al, ah          ;Get parameter byte
              call      SendCmd         ;Send parameter to the 8042.

              pop       cx
              pop       ax
              ret
SetLEDs       endp



; MyInt1C-    Every 1/4 seconds (every 4th call) this routine
;             rotates the LEDs to produce an interesting light show.

CallsPerIter  equ       4
CallCnt       byte      CallsPerIter
LEDIndex      word      LEDTable
LEDTable      byte      111b, 110b, 101b, 011b,111b, 110b, 101b, 011b
              byte      111b, 110b, 101b, 011b,111b, 110b, 101b, 011b
              byte      111b, 110b, 101b, 011b,111b, 110b, 101b, 011b
              byte      111b, 110b, 101b, 011b,111b, 110b, 101b, 011b

              byte      000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b
              byte      000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b
              byte      000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b
              byte      000b, 100b, 010b, 001b, 000b, 100b, 010b, 001b

              byte      000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b
              byte      000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b
              byte      000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b
              byte      000b, 001b, 010b, 100b, 000b, 001b, 010b, 100b

              byte      010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b
              byte      010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b
              byte      010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b
              byte      010b, 001b, 010b, 100b, 010b, 001b, 010b, 100b

              byte      000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
              byte      000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
              byte      000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
              byte      000b, 111b, 000b, 111b, 000b, 111b, 000b, 111b
TableEnd      equ       this byte

OldInt1C      dword     ?

MyInt1C       proc      far
              assume    ds:cseg

              push      ds
              push      ax
              push      bx

              mov       ax, cs
              mov       ds, ax

              dec       CallCnt
              jne       NotYet
              mov       CallCnt, CallsPerIter           ;Reset call count.
              mov       bx, LEDIndex
              mov       al, [bx]
              call      SetLEDs
```

```
                inc         bx
                cmp         bx, offset TableEnd
                jne         SetTbl
                lea         bx, LEDTable
SetTbl:         mov         LEDIndex, bx
NotYet:         pop         bx
                pop         ax
                pop         ds
                jmp         cs:OldInt1C
MyInt1C         endp


Main            proc

                mov         ax, cseg
                mov         ds, ax

                print
                byte        "LED Light Show",cr,lf
                byte        "Installing....",cr,lf,0

; Patch into the INT 1Ch interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 1Ch values directly into
; the OldInt1C variable.

                cli                         ;Turn off interrupts!
                mov         ax, 0
                mov         es, ax
                mov         ax, es:[1Ch*4]
                mov         word ptr OldInt1C, ax
                mov         ax, es:[1Ch*4 + 2]
                mov         word ptr OldInt1C+2, ax
                mov         es:[1Ch*4], offset MyInt1C
                mov         es:[1Ch*4+2], cs
                sti                         ;Okay, ints back on.


; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte        "Installed.",cr,lf,0

                mov         ah, 62h         ;Get this program's PSP
                int         21h             ; value.

                mov         dx, EndResident ;Compute size of program.
                sub         dx, bx
                mov         ax, 3100h       ;DOS TSR command.
                int         21h
Main            endp
cseg            ends

sseg            segment     para stack 'stack'
stk             db          1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

The keyboard microcontroller also sends data to the on-board microcontroller for processing and release to the system through port 60h. Most of these values are key press scan codes (up or down codes), but the keyboard transmits several other values as well. A well designed keyboard interrupt service routine should be able to handle (or at least ignore) the non-scan code values. Any particular, any program that sends commands to the keyboard needs to be able to handle the resend and acknowledge commands

that the keyboard microcontroller returns in port 60h. The keyboard microcontroller sends the following values to the system:

Table 77: Keyboard to System Transmissions

Value (hex)	Description
00	Data overrun. System sends a zero byte as the last value when the keyboard controller's internal buffer overflows.
1..58 81..D8	Scan codes for key presses. The positive values are down codes, the negative values (H.O. bit set) are up codes.
83AB	Keyboard ID code returned in response to the F2 command (PS/2 only).
AA	Returned during basic assurance test after reset. Also the up code for the left shift key.
EE	Returned by the ECHO command.
F0	Prefix to certain up codes (N/A on PS/2).
FA	Keyboard acknowledge to keyboard commands other than resend or ECHO.
FC	Basic assurance test failed (PS/2 only).
FD	Diagnostic failure (not available on PS/2).
FE	Resend. Keyboard requests the system to resend the last command.
FF	Key error (PS/2 only).

Assuming you have not disabled keyboard interrupts (see the keyboard controller command byte), any value the keyboard microcontroller sends to the system through port 60h will generate an interrupt on IRQ line one (int 9). Therefore, the keyboard interrupt service routine normally handles all the above codes. If you are patching into int 9, don't forget to send and end of interrupt (EOI) signal to the 8259A PIC at the end of your ISR code. Also, don't forget you can enable or disable the keyboard interrupt at the 8259A.

In general, your application software should *not* access the keyboard hardware directly. Doing so will probably make your software incompatible with utility software such as keyboard enhancers (keyboard macro programs), pop-up software, and other resident programs that read the keyboard or insert data into the system's type ahead buffer. Fortunately, DOS and BIOS provide an excellent set of functions to read and write keyboard data. Your programs will be much more robust if you stick to using those functions. Accessing the keyboard hardware directly should be left to keyboard ISRs and those keyboard enhancers and pop-up programs that absolutely have to talk directly to the hardware.

20.3 The Keyboard DOS Interface

MS-DOS provides several calls to read characters from the keyboard (see "MS-DOS, PC-BIOS, and File I/O" on page 699). The primary thing to note about the DOS calls is that they only return a single byte. This means that you lose the scan code information the keyboard interrupt service routine saves in the type ahead buffer.

If you press a key that has an extended code rather than an ASCII code, MS-DOS returns two keycodes. On the first call MS-DOS returns a zero value. This tells you that you must call the get character routine again. The code MS-DOS returns on the second call is the extended key code.

Note that the Standard Library routines call MS-DOS to read characters from the keyboard. Therefore, the Standard Library `getc` routine also returns extended keycodes in this manner. The `gets` and `getsm`

routines throw away any non-ASCII keystrokes since it would not be a good thing to insert zero bytes into the middle of a zero terminated string.

20.4 The Keyboard BIOS Interface

Although MS-DOS provides a reasonable set of routines to read ASCII and extended character codes from the keyboard, the PC's BIOS provides much better keyboard input facilities. Furthermore, there are lots of interesting keyboard related variables in the BIOS data area you can poke around at. In general, if you do not need the I/O redirection facilities provided by MS-DOS, reading your keyboard input using BIOS functions provides much more flexibility.

To call the MS-DOS BIOS keyboard services you use the int 16h instruction. The BIOS provides the following keyboard functions:

Table 78: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0		al- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows: bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down
3	al = 5 bh = 0, 1, 2, 3 for 1/4, 1/2, 3/4, or 1 second delay bl= 0..1Fh for 30/sec to 2/sec.		Set auto repeat rate. The bh register contains the amount of time to wait before starting the autorepeat operation, the bl register contains the autorepeat rate.
5	ch = scan code cl = ASCII code		Store keycode in buffer. This function stores the value in the cx register at the end of the type ahead buffer. Note that the scan code in ch doesn't have to correspond to the ASCII code appearing in cl. This routine will simply insert the data you provide into the system type ahead buffer.

Table 78: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
10h		al- ASCII character ah- scan code	Read extended character. Like ah=0 call, except this one passes all key codes, the ah=0 call throws away codes that are not PC/XT compatible.
11h		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Like the ah=01h call except this one does not throw away keycodes that are not PC/XT compatible (i.e., the extra keys found on the 101 key keyboard).
12h		al- shift flags ah- extended shift flags	Returns the current status of the shift flags in ax. The shift flags are defined as follows: bit 15: SysReq key pressed bit 14: Capslock key currently down bit 13: Numlock key currently down bit 12: Scroll lock key currently down bit 11: Right alt key is down bit 10:Right ctrl key is down bit 9: Left alt key is down bit 8: Left ctrl key is down bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Either alt key is down (some machines, left only) bit 2: Either ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

Note that many of these functions are not supported in every BIOS that was ever written. In fact, only the first three functions were available in the original PC. However, since the AT came along, most BIOSes have supported *at least* the functions above. Many BIOS provide extra functions, and there are many TSR applications you can buy that extend this list even farther. The following assembly code demonstrates how to write an int 16h TSR that provides all the functions above. You can easily extend this if you desire.

```
; INT16.ASM
;
; A short passive TSR that replaces the BIOS' int 16h handler.
; This routine demonstrates the function of each of the int 16h
; functions that a standard BIOS would provide.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg            segment    para public 'code'
cseg            ends

; Marker segment, to find the end of the resident section.
```

```
EndResident     segment     para public 'Resident'
EndResident     ends

                .xlist
                include     stdlib.a
                includelib stdlib.lib
                .list


byp             equ         <byte ptr>

cseg            segment     para public 'code'
                assume      cs:cseg, ds:cseg

OldInt16        dword       ?


; BIOS variables:

KbdFlags1       equ         <ds:[17h]>
KbdFlags2       equ         <ds:[18h]>
AltKpd          equ         <ds:[19h]>
HeadPtr         equ         <ds:[1ah]>
TailPtr         equ         <ds:[1ch]>
Buffer          equ         1eh
EndBuf          equ         3eh

KbdFlags3       equ         <ds:[96h]>
KbdFlags4       equ         <ds:[97h]>

incptr          macro       which
                local       NoWrap
                add         bx, 2
                cmp         bx, EndBuf
                jb          NoWrap
                mov         bx, Buffer
NoWrap:         mov         which, bx
                endm


; MyInt16-      This routine processes the int 16h function requests.
;
;               AH          Description
;               --          -----------------------------------------------
;               00h         Get a key from the keyboard, return code in AX.
;               01h         Test for available key, ZF=1 if none, ZF=0 and
;                           AX contains next key code if key available.
;               02h         Get shift status. Returns shift key status in AL.
;               03h         Set Autorepeat rate. BH=0,1,2,3 (delay time in
;                           quarter seconds), BL=0..1Fh for 30 char/sec to
;                           2 char/sec repeat rate.
;               05h         Store scan code (in CX) in the type ahead buffer.
;               10h         Get a key (same as 00h in this implementation).
;               11h         Test for key (same as 01h).
;               12h         Get extended key status. Returns status in AX.


MyInt16         proc        far
                test        ah, 0EFh            ;Check for 0h and 10h
                je          GetKey
                cmp         ah, 2               ;Check for 01h and 02h
                jb          TestKey
                je          GetStatus
                cmp         ah, 3               ;Check for AutoRpt function.
                je          SetAutoRpt
                cmp         ah, 5               ;Check for StoreKey function.
                je          StoreKey
                cmp         ah, 11h             ;Extended test key opcode.
                je          TestKey
                cmp         ah, 12h             ;Extended status call
                je          ExtStatus

; Well, it's a function we don't know about, so just return to the caller.
```

```
                iret

; If the user specified ah=0 or ah=10h, come down here (we will not
; differentiate between extended and original PC getc calls).

GetKey:         mov         ah, 11h
                int         16h                 ;See if key is available.
                je          GetKey              ;Wait for keystroke.

                push        ds
                push        bx
                mov         ax, 40h
                mov         ds, ax
                cli                             ;Critical region! Ints off.
                mov         bx, HeadPtr         ;Ptr to next character.
                mov         ax, [bx]            ;Get the character.
                incptr      HeadPtr             ;Bump up HeadPtr
                pop         bx
                pop         ds
                iret                            ;Restores interrupt flag.

; TestKey-      Checks to see if a key is available in the keyboard buffer.
;               We need to turn interrupts on here (so the kbd ISR can
;               place a character in the buffer if one is pending).
;               Generally, you would want to save the interrupt flag here.
;               But BIOS always forces interrupts on, so there may be some
;               programs out there that depend on this, so we won't "fix"
;               this problem.
;
;               Returns key status in ZF and AX. If ZF=1 then no key is
;               available and the value in AX is indeterminate. If ZF=0
;               then a key is available and AX contains the scan/ASCII
;               code of the next available key. This call does not remove
;               the next character from the input buffer.

TestKey:        sti                             ;Turn on the interrupts.
                push        ds
                push        bx
                mov         ax, 40h
                mov         ds, ax
                cli                             ;Critical region, ints off!
                mov         bx, HeadPtr
                mov         ax, [bx]            ;BIOS returns avail keycode.
                cmp         bx, TailPtr         ;ZF=1, if empty buffer
                pop         bx
                pop         ds
                sti                             ;Inst back on.
                retf        2                   ;Pop flags (ZF is important!)


; The GetStatus call simply returns the KbdFlags1 variable in AL.

GetStatus:      push        ds
                mov         ax, 40h
                mov         ds, ax
                mov         al, KbdFlags1       ;Just return Std Status.
                pop         ds
                iret


; StoreKey-     Inserts the value in CX into the type ahead buffer.

StoreKey:       push        ds
                push        bx
                mov         ax, 40h
                mov         ds, ax
                cli                             ;Ints off, critical region.
                mov         bx, TailPtr         ;Address where we can put
                push        bx                  ; next key code.
                mov         [bx], cx            ;Store the key code away.
                incptr      TailPtr             ;Move on to next entry in buf.
                cmp         bx, HeadPtr         ;Data overrun?
                jne         StoreOkay           ;If not, jump, if so
                pop         TailPtr             ; ignore key entry.
```

```
                sub     sp, 2           ;So stack matches alt path.
StoreOkay:      add     sp, 2           ;Remove junk data from stk.
                pop     bx
                pop     ds
                iret                    ;Restores interrupts.


; ExtStatus-    Retrieve the extended keyboard status and return it in
;               AH, also returns the standard keyboard status in AL.

ExtStatus:      push    ds
                mov     ax, 40h
                mov     ds, ax

                mov     ah, KbdFlags2
                and     ah, 7Fh         ;Clear final sysreq field.
                test    ah, 100b        ;Test cur sysreq bit.
                je      NoSysReq        ;Skip if it's zero.
                or      ah, 80h         ;Set final sysreq bit.
NoSysReq:
                and     ah, 0F0h        ;Clear alt/ctrl bits.
                mov     al, KbdFlags3
                and     al, 1100b       ;Grab rt alt/ctrl bits.
                or      ah, al          ;Merge into AH.
                mov     al, KbdFlags2
                and     al, 11b         ;Grab left alt/ctrl bits.
                or      ah, al          ;Merge into AH.

                mov     al, KbdFlags1   ;AL contains normal flags.
                pop     ds
                iret

; SetAutoRpt-   Sets the autorepeat rate. On entry, bh=0, 1, 2, or 3 (delay
;               in 1/4 sec before autorepeat starts) and bl=0..1Fh (repeat
;               rate, about 2:1 to 30:1 (chars:sec).

SetAutoRpt:     push    cx
                push    bx

                mov     al, 0ADh        ;Disable kbd for now.
                call    SetCmd

                and     bh, 11b         ;Force into proper range.
                mov     cl, 5
                shl     bh, cl          ;Move to final position.
                and     bl, 1Fh         ;Force into proper range.
                or      bh, bl          ;8042 command data byte.
                mov     al, 0F3h        ;8042 set repeat rate cmd.
                call    SendCmd         ;Send the command to 8042.
                mov     al, bh          ;Get parameter byte
                call    SendCmd         ;Send parameter to the 8042.

                mov     al, 0AEh        ;Reenable keyboard.
                call    SetCmd
                mov     al, 0F4h        ;Restart kbd scanning.
                call    SendCmd

                pop     bx
                pop     cx
                iret

MyInt16         endp


; SetCmd-       Sends the command byte in the AL register to the 8042
;               keyboard microcontroller chip (command register at
;               port 64h).

SetCmd          proc    near
                push    cx
                push    ax              ;Save command value.
                cli                     ;Critical region, no ints now.
```

```
; Wait until the 8042 is done processing the current command.

                xor     cx, cx          ;Allow 65,536 times thru loop.
Wait4Empty:     in      al, 64h         ;Read keyboard status register.
                test    al, 10b         ;Input buffer full?
                loopnz  Wait4Empty      ;If so, wait until empty.

; Okay, send the command to the 8042:

                pop     ax              ;Retrieve command.
                out     64h, al
                sti                     ;Okay, ints can happen again.
                pop     cx
                ret
SetCmd          endp



; SendCmd-      The following routine sends a command or data byte to the
;               keyboard data port (port 60h).

SendCmd         proc    near
                push    ds
                push    bx
                push    cx
                mov     cx, 40h
                mov     ds, cx
                mov     bx, ax          ;Save data byte

                mov     bh, 3           ;Retry cnt.
RetryLp:        cli                     ;Disable ints while accessing HW.

; Clear the Error, Acknowledge received, and resend received flags
; in KbdFlags4

                and     byte ptr KbdFlags4, 4fh

; Wait until the 8042 is done processing the current command.

                xor     cx, cx          ;Allow 65,536 times thru loop.
Wait4Empty:     in      al, 64h         ;Read keyboard status register.
                test    al, 10b         ;Input buffer full?
                loopnz  Wait4Empty      ;If so, wait until empty.

; Okay, send the data to port 60h

                mov     al, bl
                out     60h, al
                sti                     ;Allow interrupts now.

; Wait for the arrival of an acknowledgement from the keyboard ISR:

                xor     cx, cx          ;Wait a long time, if need be.
Wait4Ack:       test    byp KbdFlags4, 10 ;Acknowledge received bit.
                jnz     GotAck
                loop    Wait4Ack
                dec     bh              ;Do a retry on this guy.
                jne RetryLp

; If the operation failed after 3 retries, set the error bit and quit.

                or      byp KbdFlags4, 80h ;Set error bit.

GotAck:         pop     cx
                pop     bx
                pop     ds
                ret
SendCmd         endp



Main            proc
```

```
                mov         ax, cseg
                mov         ds, ax

                print
                byte        "INT 16h Replacement",cr,lf
                byte        "Installing....",cr,lf,0

; Patch into the INT 9 and INT 16 interrupt vectors. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 and INT 16 values directly into
; the OldInt9 and OldInt16 variables.

                cli                         ;Turn off interrupts!
                mov         ax, 0
                mov         es, ax
                mov         ax, es:[16h*4]
                mov         word ptr OldInt16, ax
                mov         ax, es:[16h*4 + 2]
                mov         word ptr OldInt16+2, ax
                mov         es:[16h*4], offset MyInt16
                mov         es:[16h*4+2], cs
                sti                         ;Okay, ints back on.


; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte        "Installed.",cr,lf,0

                mov         ah, 62h         ;Get this program's PSP
                int         21h             ; value.

                mov         dx, EndResident ;Compute size of program.
                sub         dx, bx
                mov         ax, 3100h       ;DOS TSR command.
                int         21h
Main            endp
cseg            ends

sseg            segment     para stack 'stack'
stk             db          1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

20.5 The Keyboard Interrupt Service Routine

The int 16h ISR is the interface between application programs and the keyboard. In a similar vein, the int 9 ISR is the interface between the keyboard hardware and the int 16h ISR. It is the job of the int 9 ISR to process keyboard hardware interrupts, convert incoming scan codes to scan/ASCII code combinations and place them in the typeahead buffer, and process other messages the keyboard generates.

To convert keyboard scan codes to scan/ASCII codes, the int 9 ISR must keep track of the current state of the modifier keys. When a scan code comes along, the int 9 ISR can use the `xlat` instruction to translate the scan code to an ASCII code using a table int 9 selects on the basis of the modifier flags. Another important issue is that the int 9 handler must handle special key sequences like ctrl-alt-del (reset) and PrtSc. The following assembly code provides a simple int 9 handler for the keyboard. It does not support alt-Keypad ASCII code entry or a few other minor features, but it does support almost everything you need for a keyboard interrupt service routine. Certainly it demonstrates all the techniques you need to know when programming the keyboard.

```
; INT9.ASM
;
; A short TSR to provide a driver for the keyboard hardware interrupt.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg            segment    para public 'code'
OldInt9         dword      ?
cseg            ends

; Marker segment, to find the end of the resident section.

EndResident     segment    para public 'Resident'
EndResident     ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


NumLockScan     equ        45h
ScrlLockScan    equ        46h
CapsLockScan    equ        3ah
CtrlScan        equ        1dh
AltScan         equ        38h
RShiftScan      equ        36h
LShiftScan      equ        2ah
InsScanCode     equ        52h
DelScanCode     equ        53h

; Bits for the various modifier keys

RShfBit         equ        1
LShfBit         equ        2
CtrlBit         equ        4
AltBit          equ        8
SLBit           equ        10h
NLBit           equ        20h
CLBit           equ        40h
InsBit          equ        80h


KbdFlags        equ        <byte ptr ds:[17h]>
KbdFlags2       equ        <byte ptr ds:[18h]>
KbdFlags3       equ        <byte ptr ds:[96h]>
KbdFlags4       equ        <byte ptr ds:[97h]>

byp             equ        <byte ptr>


cseg            segment    para public 'code'
                assume     ds:nothing

; Scan code translation table.
; The incoming scan code from the keyboard selects a row.
; The modifier status selects the column.
; The word at the intersection of the two is the scan/ASCII code to
; put into the PC's type ahead buffer.
; If the value fetched from the table is zero, then we do not put the
; character into the type ahead buffer.
;
;               norm    shft    ctrl    alt     num     caps    shcap   shnum

ScanXlat word   0000h, 0000h, 0000h, 0000h, 0000h, 0000h, 0000h, 0000h
         word   011bh, 011bh, 011bh, 011bh, 011bh, 011bh, 011bh, 011bh    ;ESC
         word   0231h, 0231h, 0000h, 7800h, 0231h, 0231h, 0231h, 0321h    ;1 !
```

```
                word  0332h, 0340h, 0300h, 7900h, 0332h, 0332h, 0332h, 0332h   ;2 @
                word  0433h, 0423h, 0000h, 7a00h, 0433h, 0433h, 0423h, 0423h   ;3 #
                word  0534h, 0524h, 0000h, 7b00h, 0534h, 0534h, 0524h, 0524h   ;4 $
                word  0635h, 0625h, 0000h, 7c00h, 0635h, 0635h, 0625h, 0625h   ;5 %
                word  0736h, 075eh, 071eh, 7d00h, 0736h, 0736h, 075eh, 075eh   ;6 ^

                word  0837h, 0826h, 0000h, 7e00h, 0837h, 0837h, 0826h, 0826h   ;7 &
                word  0938h, 092ah, 0000h, 7f00h, 0938h, 0938h, 092ah, 092ah   ;8 *
                word  0a39h, 0a28h, 0000h, 8000h, 0a39h, 0a39h, 0a28h, 0a28h   ;9 (
                word  0b30h, 0b29h, 0000h, 8100h, 0b30h, 0b30h, 0b29h, 0b29h   ;0 )
                word  0c2dh, 0c5fh, 0000h, 8200h, 0c2dh, 0c2dh, 0c5fh, 0c5fh   ;- _
                word  0d3dh, 0d2bh, 0000h, 8300h, 0d3dh, 0d3dh, 0d2bh, 0d2bh   ;= +
                word  0e08h, 0e08h, 0e7fh, 0000h, 0e08h, 0e08h, 0e08h, 0e08h   ;bksp
                word  0f09h, 0f00h, 0000h, 0000h, 0f09h, 0f09h, 0f00h, 0f00h   ;Tab

;               norm   shft   ctrl   alt    num   caps   shcap  shnum
                word  1071h, 1051h, 1011h, 1000h, 1071h, 1051h, 1051h, 1071h   ;Q
                word  1177h, 1057h, 1017h, 1100h, 1077h, 1057h, 1057h, 1077h   ;W
                word  1265h, 1245h, 1205h, 1200h, 1265h, 1245h, 1245h, 1265h   ;E
                word  1372h, 1352h, 1312h, 1300h, 1272h, 1252h, 1252h, 1272h   ;R
                word  1474h, 1454h, 1414h, 1400h, 1474h, 1454h, 1454h, 1474h   ;T
                word  1579h, 1559h, 1519h, 1500h, 1579h, 1559h, 1579h, 1559h   ;Y
                word  1675h, 1655h, 1615h, 1600h, 1675h, 1655h, 1675h, 1655h   ;U
                word  1769h, 1749h, 1709h, 1700h, 1769h, 1749h, 1769h, 1749h   ;I

                word  186fh, 184fh, 180fh, 1800h, 186fh, 184fh, 186fh, 184fh   ;O
                word  1970h, 1950h, 1910h, 1900h, 1970h, 1950h, 1970h, 1950h   ;P
                word  1a5bh, 1a7bh, 1a1bh, 0000h, 1a5bh, 1a5bh, 1a7bh, 1a7bh   ;[ {
                word  1b5dh, 1b7dh, 1b1dh, 0000h, 1b5dh, 1b5dh, 1b7dh, 1b7dh   ;] }
                word  1c0dh, 1c0dh, 1c0ah, 0000h, 1c0dh, 1c0dh, 1c0ah, 1c0ah   ;enter
                word  1d00h, 1d00h, 1d00h, 1d00h, 1d00h, 1d00h, 1d00h, 1d00h   ;ctrl
                word  1e61h, 1e41h, 1e01h, 1e00h, 1e61h, 1e41h, 1e61h, 1e41h   ;A
                word  1f73h, 1f5eh, 1f13h, 1f00h, 1f73h, 1f53h, 1f73h, 1f53h   ;S

;               norm   shft   ctrl   alt    num   caps   shcap  shnum
                word  2064h, 2044h, 2004h, 2000h, 2064h, 2044h, 2064h, 2044h   ;D
                word  2166h, 2146h, 2106h, 2100h, 2166h, 2146h, 2166h, 2146h   ;F
                word  2267h, 2247h, 2207h, 2200h, 2267h, 2247h, 2267h, 2247h   ;G
                word  2368h, 2348h, 2308h, 2300h, 2368h, 2348h, 2368h, 2348h   ;H
                word  246ah, 244ah, 240ah, 2400h, 246ah, 244ah, 246ah, 244ah   ;J
                word  256bh, 254bh, 250bh, 2500h, 256bh, 254bh, 256bh, 254bh   ;K
                word  266ch, 264ch, 260ch, 2600h, 266ch, 264ch, 266ch, 264ch   ;L
                word  273bh, 273ah, 0000h, 0000h, 273bh, 273bh, 273ah, 273ah   ;; :

                word  2827h, 2822h, 0000h, 0000h, 2827h, 2827h, 2822h, 2822h   ;' "
                word  2960h, 297eh, 0000h, 0000h, 2960h, 2960h, 297eh, 297eh   ;` ~
                word  2a00h, 2a00h, 2a00h, 2a00h, 2a00h, 2a00h, 2a00h, 2a00h   ;LShf
                word  2b5ch, 2b7ch, 2b1ch, 0000h, 2b5ch, 2b5ch, 2b7ch, 2b7ch   ;\ |
                word  2c7ah, 2c5ah, 2c1ah, 2c00h, 2c7ah, 2c5ah, 2c7ah, 2c5ah   ;Z
                word  2d78h, 2d58h, 2d18h, 2d00h, 2d78h, 2d58h, 2d78h, 2d58h   ;X
                word  2e63h, 2e43h, 2e03h, 2e00h, 2e63h, 2e43h, 2e63h, 2e43h   ;C
                word  2f76h, 2f56h, 2f16h, 2f00h, 2f76h, 2f56h, 2f76h, 2f56h   ;V

;               norm   shft   ctrl   alt    num   caps   shcap  shnum
                word  3062h, 3042h, 3002h, 3000h, 3062h, 3042h, 3062h, 3042h   ;B
                word  316eh, 314eh, 310eh, 3100h, 316eh, 314eh, 316eh, 314eh   ;N
                word  326dh, 324dh, 320dh, 3200h, 326dh, 324dh, 326dh, 324dh   ;M
                word  332ch, 333ch, 0000h, 0000h, 332ch, 332ch, 333ch, 333ch   ;, <
                word  342eh, 343eh, 0000h, 0000h, 342eh, 342eh, 343eh, 343eh   ;. >
                word  352fh, 353fh, 0000h, 0000h, 352fh, 352fh, 353fh, 353fh   ;/ ?
                word  3600h, 3600h, 3600h, 3600h, 3600h, 3600h, 3600h, 3600h   ;rshf
                word  372ah, 0000h, 3710h, 0000h, 372ah, 372ah, 0000h, 0000h   ;* PS

                word  3800h, 3800h, 3800h, 3800h, 3800h, 3800h, 3800h, 3800h   ;alt
                word  3920h, 3920h, 3920h, 0000h, 3920h, 3920h, 3920h, 3920h   ;spc
                word  3a00h, 3a00h, 3a00h, 3a00h, 3a00h, 3a00h, 3a00h, 3a00h   ;caps
                word  3b00h, 5400h, 5e00h, 6800h, 3b00h, 3b00h, 5400h, 5400h   ;F1
                word  3c00h, 5500h, 5f00h, 6900h, 3c00h, 3c00h, 5500h, 5500h   ;F2
                word  3d00h, 5600h, 6000h, 6a00h, 3d00h, 3d00h, 5600h, 5600h   ;F3
                word  3e00h, 5700h, 6100h, 6b00h, 3e00h, 3e00h, 5700h, 5700h   ;F4
                word  3f00h, 5800h, 6200h, 6c00h, 3f00h, 3f00h, 5800h, 5800h   ;F5

;               norm   shft   ctrl   alt    num   caps   shcap  shnum
                word  4000h, 5900h, 6300h, 6d00h, 4000h, 4000h, 5900h, 5900h   ;F6
```

```
        word  4100h, 5a00h, 6400h, 6e00h, 4100h, 4100h, 5a00h, 5a00h    ;F7
        word  4200h, 5b00h, 6500h, 6f00h, 4200h, 4200h, 5b00h, 5b00h    ;F8
        word  4300h, 5c00h, 6600h, 7000h, 4300h, 4300h, 5c00h, 5c00h    ;F9
        word  4400h, 5d00h, 6700h, 7100h, 4400h, 4400h, 5d00h, 5d00h    ;F10
        word  4500h, 4500h, 4500h, 4500h, 4500h, 4500h, 4500h, 4500h    ;num
        word  4600h, 4600h, 4600h, 4600h, 4600h, 4600h, 4600h, 4600h    ;scrl
        word  4700h, 4737h, 7700h, 0000h, 4737h, 4700h, 4737h, 4700h    ;home

        word  4800h, 4838h, 0000h, 0000h, 4838h, 4800h, 4838h, 4800h    ;up
        word  4900h, 4939h, 8400h, 0000h, 4939h, 4900h, 4939h, 4900h    ;pgup
        word  4a2dh, 4a2dh, 0000h, 0000h, 4a2dh, 4a2dh, 4a2dh, 4a2dh    ;-
        word  4b00h, 4b34h, 7300h, 0000h, 4b34h, 4b00h, 4b34h, 4b00h    ;left
        word  4c00h, 4c35h, 0000h, 0000h, 4c35h, 4c00h, 4c35h, 4c00h    ;Center
        word  4d00h, 4d36h, 7400h, 0000h, 4d36h, 4d00h, 4d36h, 4d00h    ;right
        word  4e2bh, 4e2bh, 0000h, 0000h, 4e2bh, 4e2bh, 4e2bh, 4e2bh    ;+
        word  4f00h, 4f31h, 7500h, 0000h, 4f31h, 4f00h, 4f31h, 4f00h    ;end

;               norm   shft   ctrl   alt    num   caps   shcap  shnum
        word  5000h, 5032h, 0000h, 0000h, 5032h, 5000h, 5032h, 5000h    ;down
        word  5100h, 5133h, 7600h, 0000h, 5133h, 5100h, 5133h, 5100h    ;pgdn
        word  5200h, 5230h, 0000h, 0000h, 5230h, 5200h, 5230h, 5200h    ;ins
        word  5300h, 532eh, 0000h, 0000h, 532eh, 5300h, 532eh, 5300h    ;del
        word  0,0,0,0,0,0,0,0                                           ; --
        word  0,0,0,0,0,0,0,0                                           ; --
        word  0,0,0,0,0,0,0,0                                           ; --
        word  5700h, 0000h, 0000h, 0000h, 5700h, 5700h, 0000h, 0000h    ;F11

        word  5800h, 0000h, 0000h, 0000h, 5800h, 5800h, 0000h, 0000h    ;F12



;******************************************************************************
;
; AL contains keyboard scan code.

PutInBuffer   proc near
              push ds
              push bx

              mov bx, 40h                ;Point ES at the BIOS
              mov ds, bx                 ; variables.

; If the current scan code is E0 or E1, we need to take note of this fact
; so that we can properly process cursor keys.

              cmp       al, 0e0h
              jne       TryE1
              or        KbdFlags3, 10b    ;Set E0 flag
              and       KbdFlags3, 0FEh   ;Clear E1 flag
              jmp       Done

TryE1:        cmp       al, 0e1h
              jne       DoScan
              or        KbdFlags3, 1      ;Set E1 flag
              and       KbdFlags3, 0FDh   ;Clear E0 Flag
              jmp       Done


; Before doing anything else, see if this is Ctrl-Alt-Del:

DoScan:       cmp       al, DelScanCode
              jnz       TryIns
              mov       bl, KbdFlags
              and       bl, AltBit or CtrlBit ;Alt = bit 3, ctrl = bit 2
              cmp       bl, AltBit or CtrlBit
              jne       DoPIB
              mov       word ptr ds:[72h], 1234h ;Warm boot flag.
              jmp       dword ptr cs:RebootAdrs  ;REBOOT Computer

RebootAdrs    dword     0ffff0000h        ;Reset address.


; Check for the INS key here. This one needs to toggle the ins bit
; in the keyboard flags variables.
```

```
TryIns:       cmp       al, InsScanCode
              jne       TryInsUp
              or        KbdFlags2, InsBit           ;Note INS is down.
              jmp       doPIB                       ;Pass on INS key.

TryInsUp:     cmp       al, InsScanCode+80h         ;INS up scan code.
              jne       TryLShiftDn
              and       KbdFlags2, not InsBit       ;Note INS is up.
              xor       KbdFlags, InsBit            ;Toggle INS bit.
              jmp       QuitPIB

; Handle the left and right shift keys down here.

TryLShiftDn:  cmp       al, LShiftScan
              jne       TryLShiftUp
              or        KbdFlags, LShfBit           ;Note that the left
              jmp       QuitPIB                     ; shift key is down.

TryLShiftUp:  cmp       al, LShiftScan+80h
              jne       TryRShiftDn
              and       KbdFlags, not LShfBit       ;Note that the left
              jmp       QuitPIB                     ; shift key is up.


TryRShiftDn:  cmp       al, RShiftScan
              jne       TryRShiftUp
              or        KbdFlags, RShfBit           ;Right shf is down.
              jmp       QuitPIB

TryRShiftUp:  cmp       al, RShiftScan+80h
              jne       TryAltDn
              and       KbdFlags, not RShfBit       ;Right shf is up.
              jmp       QuitPIB

; Handle the ALT key down here.

TryAltDn:     cmp       al, AltScan
              jne       TryAltUp
              or        KbdFlags, AltBit            ;Alt key is down.
GotoQPIB:     jmp       QuitPIB

TryAltUp:     cmp       al, AltScan+80h
              jne       TryCtrlDn
              and       KbdFlags, not AltBit        ;Alt key is up.
              jmp       DoPIB


; Deal with the control key down here.

TryCtrlDn:    cmp       al, CtrlScan
              jne       TryCtrlUp
              or        KbdFlags, CtrlBit           ;Ctrl key is down.
              jmp       QuitPIB

TryCtrlUp:    cmp       al, CtrlScan+80h
              jne       TryCapsDn
              and       KbdFlags, not CtrlBit       ;Ctrl key is up.
              jmp       QuitPIB

; Deal with the CapsLock key down here.

TryCapsDn:    cmp       al, CapsLockScan
              jne       TryCapsUp
              or        KbdFlags2, CLBit            ;Capslock is down.
              xor       KbdFlags, CLBit             ;Toggle capslock.
              jmp       QuitPIB

TryCapsUp:    cmp       al, CapsLockScan+80h
              jne       TrySLDn
              and       KbdFlags2, not CLBit        ;Capslock is up.
              call      SetLEDs
              jmp       QuitPIB
```

```
; Deal with the Scroll Lock key down here.

TrySLDn:        cmp         al, ScrlLockScan
                jne         TrySLUp
                or          KbdFlags2, SLBit                ;Scrl lock is down.
                xor         KbdFlags, SLBit                 ;Toggle scrl lock.
                jmp         QuitPIB

TrySLUp:        cmp         al, ScrlLockScan+80h
                jne         TryNLDn
                and         KbdFlags2, not SLBit            ;Scrl lock is up.
                call        SetLEDs
                jmp         QuitPIB

; Handle the NumLock key down here.


TryNLDn:        cmp         al, NumLockScan
                jne         TryNLUp
                or          KbdFlags2, NLBit                ;Numlock is down.
                xor         KbdFlags, NLBit                 ;Toggle numlock.
                jmp         QuitPIB

TryNLUp:        cmp         al, NumLockScan+80h
                jne         DoPIB
                and         KbdFlags2, not NLBit            ;Numlock is up.
                call        SetLEDs
                jmp         QuitPIB



; Handle all the other keys here:

DoPIB:          test        al, 80h                        ;Ignore other up keys.
                jnz         QuitPIB

; If the H.O. bit is set at this point, we'd best only have a zero in AL.
; Otherwise, this is an up code which we can safely ignore.

                call        Convert
                test        ax, ax                          ;Chk for bad code.
                je          QuitPIB

PutCharInBuf:   push        cx
                mov         cx, ax
                mov         ah, 5                           ;Store scan code into
                int         16h                             ; type ahead buffer.
                pop         cx

QuitPIB:        and         KbdFlags3, 0FCh                 ;E0, E1 not last code.

Done:           pop bx
                pop ds
                ret
PutInBuffer     endp



;*****************************************************************************
;
; Convert-      AL contains a PC Scan code. Convert it to an ASCII char/Scan
;               code pair and return the result in AX. This code assumes
;               that DS points at the BIOS variable space (40h).

Convert         proc        near
                push        bx

                test        al, 80h             ;See if up code
                jz          DownScanCode
                mov         ah, al
                mov         al, 0
                jmp         CSDone
```

```
; Okay, we've got a down key. But before going on, let's see if we've
; got an ALT-Keypad sequence.

DownScanCode:  mov       bh, 0
               mov       bl, al
               shl       bx, 1            ;Multiply by eight to compute
               shl       bx, 1            ; row index index the scan
               shl       bx, 1            ; code xlat table

; Compute modifier index as follows:
;
;       if alt then modifier = 3

               test      KbdFlags, AltBit
               je        NotAlt
               add       bl, 3
               jmp       DoConvert

;       if ctrl, then modifier = 2

NotAlt:        test      KbdFlags, CtrlBit
               je        NotCtrl
               add       bl, 2
               jmp       DoConvert

; Regardless of the shift setting, we've got to deal with numlock
; and capslock. Numlock is only a concern if the scan code is greater
; than or equal to 47h. Capslock is only a concern if the scan code
; is less than this.

NotCtrl:       cmp       al, 47h
               jb        DoCapsLk
               test      KbdFlags, NLBit                 ;Test Numlock bit
               je        NoNumLck
               test      KbdFlags, LShfBit or RShfBit    ;Check l/r shift.
               je        NumOnly
               add       bl, 7                           ;Numlock and shift.
               jmp       DoConvert

NumOnly:       add       bl, 4                           ;Numlock only.
               jmp       DoConvert

; If numlock is not active, see if a shift key is:

NoNumLck:      test      KbdFlags, LShfBit or RShfBit    ;Check l/r shift.
               je        DoConvert                       ;normal if no shift.
               add       bl, 1
               jmp       DoConvert

; If the scan code's value is below 47h, we need to check for capslock.

DoCapsLk:      test      KbdFlags, CLBit                 ;Chk capslock bit
               je        DoShift
               test      KbdFlags, LShfBit or RShfBit    ;Chk for l/r shift
               je        CapsOnly
               add       bl, 6                           ;Shift and capslock.
               jmp       DoConvert

CapsOnly:      add       bl, 5                           ;Capslock
               jmp       DoConvert

; Well, nothing else is active, check for just a shift key.

DoShift:       test      KbdFlags, LShfBit or RShfBit    ;l/r shift.
               je        DoConvert
               add       bl, 1                           ;Shift

DoConvert:     shl       bx, 1                           ;Word array
               mov       ax, ScanXlat[bx]
CSDone:        pop       bx
               ret
Convert        endp
```

```
; SetCmd-     Sends the command byte in the AL register to the 8042
;             keyboard microcontroller chip (command register at
;             port 64h).

SetCmd        proc       near
              push       cx
              push       ax              ;Save command value.
              cli                        ;Critical region, no ints now.

; Wait until the 8042 is done processing the current command.

              xor        cx, cx          ;Allow 65,536 times thru loop.
Wait4Empty:   in         al, 64h         ;Read keyboard status register.
              test       al, 10b         ;Input buffer full?
              loopnz     Wait4Empty      ;If so, wait until empty.

; Okay, send the command to the 8042:

              pop        ax              ;Retrieve command.
              out        64h, al
              sti                        ;Okay, ints can happen again.
              pop        cx
              ret
SetCmd        endp



; SendCmd-    The following routine sends a command or data byte to the
;             keyboard data port (port 60h).

SendCmd       proc       near
              push       ds
              push       bx
              push       cx
              mov        cx, 40h
              mov        ds, cx
              mov        bx, ax          ;Save data byte

              mov        bh, 3           ;Retry cnt.
RetryLp:      cli                        ;Disable ints while accessing HW.

; Clear the Error, Acknowledge received, and resend received flags
; in KbdFlags4

              and        byte ptr KbdFlags4, 4fh

; Wait until the 8042 is done processing the current command.

              xor        cx, cx             ;Allow 65,536 times thru loop.
Wait4Empty:   in         al, 64h            ;Read keyboard status register.
              test       al, 10b            ;Input buffer full?
              loopnz     Wait4Empty         ;If so, wait until empty.

; Okay, send the data to port 60h

              mov        al, bl
              out        60h, al
              sti                           ;Allow interrupts now.

; Wait for the arrival of an acknowledgement from the keyboard ISR:

              xor        cx, cx             ;Wait a long time, if need be.
Wait4Ack:     test       byp KbdFlags4,10h ;Acknowledge received bit.
              jnz        GotAck
              loop       Wait4Ack
              dec        bh                 ;Do a retry on this guy.
              jne RetryLp

; If the operation failed after 3 retries, set the error bit and quit.

              or         byp KbdFlags4,80h ;Set error bit.
```

```
GotAck:         pop         cx
                pop         bx
                pop         ds
                ret
SendCmd         endp



; SetLEDs-      Updates the KbdFlags4 LED bits from the KbdFlags
;               variable and then transmits new flag settings to
;               the keyboard.

SetLEDs         proc        near
                push        ax
                push        cx
                mov         al, KbdFlags
                mov         cl, 4
                shr         al, cl
                and         al, 111b
                and         KbdFlags4, 0F8h   ;Clear LED bits.
                or          KbdFlags4, al     ;Mask in new bits.
                mov         ah, al            ;Save LED bits.

                mov         al, 0ADh          ;Disable kbd for now.
                call        SetCmd

                mov         al, 0EDh          ;8042 set LEDs cmd.
                call        SendCmd           ;Send the command to 8042.
                mov         al, ah            ;Get parameter byte
                call        SendCmd           ;Send parameter to the 8042.

                mov         al, 0AEh          ;Reenable keyboard.
                call        SetCmd
                mov         al, 0F4h          ;Restart kbd scanning.
                call        SendCmd
                pop         cx
                pop         ax
                ret
SetLEDs         endp


; MyInt9-       Interrupt service routine for the keyboard hardware
;               interrupt.

MyInt9          proc        far
                push        ds
                push        ax
                push        cx

                mov         ax, 40h
                mov         ds, ax

                mov         al, 0ADh          ;Disable keyboard
                call        SetCmd
                cli                           ;Disable interrupts.
                xor         cx, cx
Wait4Data:      in          al, 64h           ;Read kbd status port.
                test        al, 10b           ;Data in buffer?
                loopz       Wait4Data         ;Wait until data available.
                in          al, 60h           ;Get keyboard data.
                cmp         al, 0EEh          ;Echo response?
                je          QuitInt9
                cmp         al, 0FAh          ;Acknowledge?
                jne         NotAck
                or          KbdFlags4, 10h    ;Set ack bit.
                jmp         QuitInt9

NotAck:         cmp         al, 0FEh          ;Resend command?
                jne         NotResend
                or          KbdFlags4, 20h    ;Set resend bit.
                jmp         QuitInt9

; Note: other keyboard controller commands all have their H.O. bit set
```

```
; and the PutInBuffer routine will ignore them.

NotResend:      call        PutInBuffer         ;Put in type ahead buffer.

QuitInt9:       mov         al, 0AEh            ;Reenable the keyboard
                call        SetCmd

                mov         al, 20h             ;Send EOI (end of interrupt)
                out         20h, al             ; to the 8259A PIC.
                pop         cx
                pop         ax
                pop         ds
                iret
MyInt9          endp



Main            proc
                assume      ds:cseg

                mov         ax, cseg
                mov         ds, ax

                print
                byte        "INT 9 Replacement",cr,lf
                byte        "Installing....",cr,lf,0

; Patch into the INT 9 interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 value directly into
; the OldInt9 variable.

                cli                             ;Turn off interrupts!
                mov         ax, 0
                mov         es, ax
                mov         ax, es:[9*4]
                mov         word ptr OldInt9, ax
                mov         ax, es:[9*4 + 2]
                mov         word ptr OldInt9+2, ax
                mov         es:[9*4], offset MyInt9
                mov         es:[9*4+2], cs
                sti                             ;Okay, ints back on.


; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte        "Installed.",cr,lf,0

                mov         ah, 62h             ;Get this program's PSP
                int         21h                 ; value.

                mov         dx, EndResident     ;Compute size of program.
                sub         dx, bx
                mov         ax, 3100h           ;DOS TSR command.
                int         21h
Main            endp
cseg            ends

sseg            segment     para stack 'stack'
stk             byte        1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       db          16 dup (?)
zzzzzzseg       ends
                end         Main
```

20.6 Patching into the INT 9 Interrupt Service Routine

For many programs, such as pop-up programs or keyboard enhancers, you may need to intercept certain "hot keys" and pass all remaining scan codes through to the default keyboard interrupt service routine. You can insert an int 9 interrupt service routine into an interrupt nine chain just like any other interrupt. When the keyboard interrupts the system to send a scan code, your interrupt service routine can read the scan code from port 60h and decide whether to process the scan code itself or pass control on to some other int 9 handler. The following program demonstrates this principle; it deactivates the ctrl-alt-del reset function on the keyboard by intercepting and throwing away delete scan codes when the ctrl and alt bits are set in the keyboard flags byte.

```
; NORESET.ASM
;
; A short TSR that patches the int 9 interrupt and intercepts the
; ctrl-alt-del keystroke sequence.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg            segment    para public 'code'
OldInt9         dword      ?
cseg            ends

; Marker segment, to find the end of the resident section.

EndResident     segment    para public 'Resident'
EndResident     ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


DelScanCode     equ        53h

; Bits for the various modifier keys

CtrlBit         equ        4
AltBit          equ        8


KbdFlags        equ        <byte ptr ds:[17h]>



cseg            segment    para public 'code'
                assume     ds:nothing


; SetCmd-       Sends the command byte in the AL register to the 8042
;               keyboard microcontroller chip (command register at
;               port 64h).

SetCmd          proc       near
                push       cx
                push       ax              ;Save command value.
                cli                        ;Critical region, no ints now.

; Wait until the 8042 is done processing the current command.

                xor        cx, cx          ;Allow 65,536 times thru loop.
Wait4Empty:     in         al, 64h         ;Read keyboard status register.
```

```
                test    al, 10b         ;Input buffer full?
                loopnz  Wait4Empty      ;If so, wait until empty.

; Okay, send the command to the 8042:

                pop     ax              ;Retrieve command.
                out     64h, al
                sti                     ;Okay, ints can happen again.
                pop     cx
                ret
SetCmd          endp


; MyInt9-       Interrupt service routine for the keyboard hardware
;               interrupt. Tests to see if the user has pressed a
;               DEL key. If not, it passes control on to the original
;               int 9 handler. If so, it first checks to see if the
;               alt and ctrl keys are currently down; if not, it passes
;               control to the original handler. Otherwise it eats the
;               scan code and doesn't pass the DEL through.

MyInt9          proc    far
                push    ds
                push    ax
                push    cx

                mov     ax, 40h
                mov     ds, ax

                mov     al, 0ADh        ;Disable keyboard
                call    SetCmd
                cli                     ;Disable interrupts.
                xor     cx, cx
Wait4Data:      in      al, 64h         ;Read kbd status port.
                test    al, 10b         ;Data in buffer?
                loopz   Wait4Data       ;Wait until data available.

                in      al, 60h         ;Get keyboard data.
                cmp     al, DelScanCode ;Is it the delete key?
                jne     OrigInt9
                mov     al, KbdFlags    ;Okay, we've got DEL, is
                and     al, AltBit or CtrlBit ; ctrl+alt down too?
                cmp     al, AltBit or CtrlBit
                jne     OrigInt9

; If ctrl+alt+DEL is down, just eat the DEL code and don't pass it through.

                mov     al, 0AEh        ;Reenable the keyboard
                call    SetCmd

                mov     al, 20h         ;Send EOI (end of interrupt)
                out     20h, al         ; to the 8259A PIC.
                pop     cx
                pop     ax
                pop     ds
                iret

; If ctrl and alt aren't both down, pass DEL on to the original INT 9
; handler routine.

OrigInt9:       mov     al, 0AEh        ;Reenable the keyboard
                call    SetCmd

                pop     cx
                pop     ax
                pop     ds
                jmp     cs:OldInt9
MyInt9          endp



Main            proc
                assume  ds:cseg
```

```
                mov     ax, cseg
                mov     ds, ax

                print
                byte    "Ctrl-Alt-Del Filter",cr,lf
                byte    "Installing....",cr,lf,0

; Patch into the INT 9 interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 value directly into
; the OldInt9 variable.

                cli                     ;Turn off interrupts!
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[9*4]
                mov     word ptr OldInt9, ax
                mov     ax, es:[9*4 + 2]
                mov     word ptr OldInt9+2, ax
                mov     es:[9*4], offset MyInt9
                mov     es:[9*4+2], cs
                sti                     ;Okay, ints back on.


; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte    "Installed.",cr,lf,0

                mov     ah, 62h         ;Get this program's PSP
                int     21h             ; value.

                mov     dx, EndResident ;Compute size of program.
                sub     dx, bx
                mov     ax, 3100h       ;DOS TSR command.
                int     21h
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

20.7 Simulating Keystrokes

At one point or another you may want to write a program that passes keystrokes on to another application. For example, you might want to write a keyboard macro TSR that lets you capture certain keys on the keyboard and send a sequence of keys through to some underlying application. Perhaps you'll want to program an entire string of characters on a normally unused keyboard sequence (e.g., ctrl-up or ctrl--down). In any case, your program will use some technique to pass characters to a foreground application. There are three well-known techniques for doing this: store the scan/ASCII code directly in the keyboard buffer, use the 80x86 *trace* flag to simulate `in al, 60h` instructions, or program the on-board 8042 microcontroller to transmit the scan code for you. The next three sections describe these techniques in detail.

20.7.1 Stuffing Characters in the Type Ahead Buffer

Perhaps the easiest way to insert keystrokes into an application is to insert them directly into the system's type ahead buffer. Most modern BIOSes provide an int 16h function to do this (see "The Keyboard

BIOS Interface" on page 1168). Even if your system does not provide this function, it is easy to write your own code to insert data in the system type ahead buffer; or you can copy the code from the int 16h handler provided earlier in this chapter.

The nice thing about this approach is that you can deal directly with ASCII characters (at least, for those key sequences that are ASCII). You do not have to worry about sending shift up and down codes around the scan code for tn "A" so you can get an upper case "A", you need only insert 1E41h into the buffer. In fact, most programs ignore the scan code, so you can simply insert 0041h into the buffer and almost any application will accept the funny scan code of zero.

The major drawback to the buffer insertion technique is that many (popular) applications bypass DOS and BIOS when reading the keyboard. Such programs go directly to the keyboard's port (60h) to read their data. As such, shoving scan/ASCII codes into the type ahead buffer will have no effect. Ideally, you would like to stuff a scan code directly into the keyboard controller chip and have it return that scan code as though someone actually pressed that key. Unfortunately, there is no universally compatible way to do this. However, there are some close approximations, keep reading...

20.7.2 Using the 80x86 Trace Flag to Simulate IN AL, 60H Instructions

One way to deal with applications that access the keyboard hardware directly is to *simulate* the 80x86 instruction set. For example, suppose we were able to take control of the int 9 interrupt service routine and execute each instruction under our control. We could choose to let all instructions *except* the `in` instruction execute normally. Upon encountering an `in` instruction (that the keyboard ISR uses to read the keyboard data), we check to see if it is accessing port 60h. If so, we simply load the `al` register with the desired scan code rather than actually execute the `in` instruction. It is also important to check for the `out` instruction, since the keyboard ISR will want to send and EOI signal to the 8259A PIC after reading the keyboard data, we can simply ignore `out` instructions that write to port 20h.

The only difficult part is telling the 80x86 to pass control to our routine when encountering certain instructions (like `in` and `out`) and to execute other instructions normally. While this is not directly possible in real mode[7], there is a close approximation we can make. The 80x86 CPUs provide a *trace* flag that generates an exception after the execution of each instruction. Normally, debuggers use the trace flag to single step through a program. However, by writing our own exception handler for the trace exception, we can gain control of the machine between the execution of every instruction. Then, we can look at the opcode of the next instruction to execute. If it is not an `in` or `out` instruction, we can simply return and execute the instruction normally. If it is an `in` or `out` instruction, we can determine the I/O address and decide whether to simulate or execute the instruction.

In addition to the `in` and `out` instructions, we will need to simulate any `int` instructions we find as well. The reason is because the `int` instruction pushes the flags on the stack and then clears the trace bit in the flags register. This means that the interrupt service routine associated with that `int` instruction would execute normally and we would miss any `in` or `out` instructions appearing therein. However, it is easy to simulate the `int` instruction, leaving the trace flag enabled, so we will add `int` to our list of instructions to interpret.

The only problem with this approach is that it is slow. Although the trace trap routine will only execute a few instructions on each call, it does so for every instruction in the int 9 interrupt service routine. As a result, during simulation, the interrupt service routine will run 10 to 20 times slower than the real code would. This generally isn't a problem because most keyboard interrupt service routines are very short. However, you might encounter an application that has a large internal int 9 ISR and this method would noticeably slow the program. However, for most applications this technique works just fine and no one will notice any performance loss while they are typing away (slowly) at the keyboard.

7. It is possible to trap I/O instructions when running in protected mode.

The following assembly code provides a short example of a trace exception handler that simulates keystrokes in this fashion:

```
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

cseg            segment    para public 'code'
                assume     ds:nothing

; ScanCode must be in the Code segment.

ScanCode        byte       0



;*****************************************************************************
;
; KbdSim- Passes the scan code in AL through the keyboard controller
; using the trace flag. The way this works is to turn on the
; trace bit in the flags register. Each instruction then causes a trace
; trap. The (installed) trace handler then looks at each instruction to
; handle IN, OUT, INT, and other special instructions. Upon encountering
; an IN AL, 60 (or equivalent) this code simulates the instruction and
; returns the specified scan code rather than actually executing the IN
; instruction. Other instructions need special treatment as well. See
; the code for details. This code is pretty good at simulating the hardware,
; but it runs fairly slow and has a few compatibility problems.


KbdSim          proc       near

                pushf
                push       es
                push       ax
                push       bx


                xor        bx, bx            ;Point es at int vector tbl
                mov        es, bx            ; (to simulate INT 9).
                cli                          ;No interrupts for now.
                mov        cs:ScanCode, al   ;Save output scan code.

                push       es:[1*4]          ;Save current INT 1 vector
                push       es:2[1*4]         ; so we can restore it later.



; Point the INT 1 vector at our INT 1 handler:

                mov        word ptr es:[1*4], offset MyInt1
                mov        word ptr es:[1*4 + 2], cs



; Turn on the trace trap (bit 8 of flags register):

                pushf
                pop        ax
                or         ah, 1
                push       ax
                popf


; Simulate an INT 9 instruction. Note: cannot actually execute INT 9 here
; since INT instructions turn off the trace operation.


                pushf
                call       dword ptr es:[9*4]
```

```
; Turn off the trace operation:

                pushf
                pop         ax
                and         ah, 0feh          ;Clear trace bit.
                push        ax
                popf


; Disable trace operation.


                pop         es:[1*4 + 2]      ;Restore previous INT 1
                pop         es:[1*4]          ; handler.


; Okay, we're done. Restore registers and return.

VMDone:         pop         bx
                pop         ax
                pop         es
                popf
                ret
KbdSim          endp



;-----------------------------------------------------------------------------
;
; MyInt1- Handles the trace trap (INT 1). This code looks at the next
; opcode to determine if it is one of the special opcodes we have to
; handle ourselves.


MyInt1          proc        far
                push        bp
                mov         bp, sp            ;Gain access to return adrs via BP.
                push        bx
                push        ds

; If we get down here, it's because this trace trap is directly due to
; our having punched the trace bit. Let's process the trace trap to
; simulate the 80x86 instruction set.
;
; Get the return address into DS:BX

NextInstr:      lds         bx, 2[bp]

; The following is a special case to quickly eliminate most opcodes and
; speed up this code by a tiny amount.

                cmp         byte ptr [bx], 0cdh ;Most opcodes are less than
                jnb         NotSimple         ; 0cdh, hence we quickly
                pop         ds                ; return back to the real
                pop         bx                ; program.
                pop         bp
                iret

NotSimple:      je          IsIntInstr        ;If it's an INT instruction.

                mov         bx, [bx]          ;Get current instruction's opcode.
                cmp         bl, 0e8h          ;CALL opcode
                je          ExecInstr
                jb          TryInOut0

                cmp         bl, 0ech          ;IN al, dx instr.
                je          MayBeIn60
                cmp         bl, 0eeh          ;OUT dx, al instr.
                je          MayBeOut20
                pop         ds                ;A normal instruction if we get
                pop         bx                ; down here.
                pop         bp
                iret
```

```
TryInOut0:      cmp         bx, 60e4h       ;IN al, 60h instr.
                je          IsINAL60
                cmp         bx, 20e6h       ;out 20, al instr.
                je          IsOut20

; If it wasn't one of our magic instructions, execute it and continue.

ExecInstr:      pop         ds
                pop         bx
                pop         bp
                iret

; If this instruction is IN AL, DX we have to look at the value in DX to
; determine if it's really an IN AL, 60h instruction.

MayBeIn60:      cmp         dx, 60h
                jne         ExecInstr
                inc         word ptr 2[bp]    ;Skip over this 1 byte instr.
                mov         al, cs:ScanCode
                jmp         NextInstr

; If this is an IN AL, 60h instruction, simulate it by loading the current
; scan code into AL.

IsInAL60:       mov         al, cs:ScanCode
                add         word ptr 2[bp], 2 ;Skip over this 2-byte instr.
                jmp         NextInstr


; If this instruction is OUT DX, AL we have to look at DX to see if we're
; outputting to location 20h (8259).

MayBeOut20:     cmp         dx, 20h
                jne         ExecInstr
                inc         word ptr 2[bp]    ;Skip this 1 byte instruction.
                jmp         NextInstr

; If this is an OUT 20h, al instruction, simply skip over it.

IsOut20:        add         word ptr 2[bp], 2 ;Skip instruction.
                jmp         NextInstr


; IsIntInstr- Execute this code if it's an INT instruction.
;
; The problem with the INT instructions is that they reset the trace bit
; upon execution. For certain guys (see above) we can't have that.
;
; Note: at this point the stack looks like the following:
;
;       flags
;
;       rtn cs -+
;               |
;       rtn ip  +-- Points at next instr the CPU will execute.
;       bp
;       bx
;       ds
;
; We need to simulate the appropriate INT instruction by:
;
;       (1)   adding two to the return address on the stack (so it returns
;             beyond the INT instruction.
;       (2)   pushing the flags onto the stack.
;       (3)   pushing a phony return address onto the stack which simulates
;             the INT 1 interrupt return address but which "returns" us to
;             the specified interrupt vector handler.
;
; All this results in a stack which looks like the following:
;
;       flags
;
;       rtn cs -+
```

```
;               |
;       rtn ip  +-- Points at next instr beyond the INT instruction.
;
;       flags   --- Bogus flags to simulate those pushed by INT instr.
;
;       rtn cs -+
;               |
;       rtn ip  +-- "Return address" which points at the ISR for this INT.
;       bp
;       bx
;       ds


IsINTInstr:   add       word ptr 2[bp], 2 ;Bump rtn adrs beyond INT instr.
              mov       bl, 1[bx]
              mov       bh, 0
              shl       bx, 1             ;Multiply by 4 to get vector
              shl       bx, 1             ; address.

              push      [bp-0]            ;Get and save BP
              push      [bp-2]            ;Get and save BX.
              push      [bp-4]            ;Get and save DS.

              push      cx
              xor       cx, cx            ;Point DS at interrupt
              mov       ds, cx            ; vector table.

              mov       cx, [bp+6]        ;Get original flags.
              mov       [bp-0], cx        ;Save as pushed flags.

              mov       cx, ds:2[bx]      ;Get vector and use it as
              mov       [bp-2], cx        ; the return address.
              mov       cx, ds:[bx]
              mov       [bp-4], cx

              pop       cx
              pop       ds
              pop       bx
              pop       bp
              iret
;
MyInt1        endp



; Main program - Simulates some keystrokes to demo the above code.

Main          proc

              mov       ax, cseg
              mov       ds, ax

              print
              byte      "Simulating keystrokes via Trace Flag",cr,lf
              byte      "This program places 'DIR' in the keyboard buffer"
              byte      cr,lf,0

              mov       al, 20h           ;"D" down scan code
              call      KbdSim
              mov       al, 0a0h          ;"D" up scan code
              call      KbdSim

              mov       al, 17h           ;"I" down scan code
              call      KbdSim
              mov       al, 97h           ;"I" up scan code
              call      KbdSim

              mov       al, 13h           ;"R" down scan code
              call      KbdSim
              mov       al, 93h           ;"R" up scan code
              call      KbdSim

              mov       al, 1Ch           ;Enter down scan code
```

```
                call    KbdSim
                mov     al, 9Ch         ;Enter up scan code
                call    KbdSim



                ExitPgm
Main            endp


cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

20.7.3 Using the 8042 Microcontroller to Simulate Keystrokes

Although the trace flag based "keyboard stuffer" routine works with most software that talks to the hardware directly, it still has a few problems. Specifically, it doesn't work at all with programs that operate in protected mode via a "DOS Extender" library (programming libraries that let programmers access more than one megabyte of memory while running under DOS). The last technique we will look at is to program the on-board 8042 keyboard microcontroller to transmit a keystroke for us. There are two ways to do this: the PS/2 way and the hard way.

The PS/2's microcontroller includes a command specifically designed to return user programmable scan codes to the system. By writing a 0D2h byte to the controller command port (64h) and a scan code byte to port 60h, you can force the controller to return that scan code as though the user pressed a key on the keyboard. See "The Keyboard Hardware Interface" on page 1159 for more details.

Using this technique provides the most compatible (with existing software) way to return scan codes to an application. Unfortunately, this trick only works on machines that have keyboard controllers that are compatible with the PS/2's; this is not the majority of machines out there. However, if you are writing code for PS/2s or compatibles, this is the best way to go.

The keyboard controller on the PC/AT and most other PC compatible machines does not support the 0D2h command. Nevertheless, there is a sneaky way to force the keyboard controller to transmit a scan code, if you're willing to break a few rules. This trick may not work on all machines (indeed, there are many machines on which this trick is known to fail), but it does provide a workaround on a large number of PC compatible machines.

The trick is simple. Although the PC's keyboard controller doesn't have a command to return a byte you send it, it does provide a command to return the keyboard controller command byte (KCCB). It also provides another command to write a value to the KCCB. So by writing a value to the KCCB and then issuing the read KCCB command, we can trick the system into returning a user programmable code. Unfortunately, the KCCB contains some undefined reserved bits that have different meanings on different brands of keyboard microcontroller chips. That is the main reason this technique doesn't work with all machines. The following assembly code demonstrates how to use the PS/2 and PC keyboard controller stuffing methods:

```
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

cseg            segment    para public 'code'
```

```
                assume  ds:nothing

;*****************************************************************************
;
; PutInATBuffer-
;
; The following code sticks the scan code into the AT-class keyboard
; microcontroller chip and asks it to send the scan code back to us
; (through the hardware port).
;
; The AT keyboard controller:
;
; Data port is at I/O address 60h
; Status port is at I/O address 64h (read only)
; Command port is at I/O address 64h (write only)
;
; The controller responds to the following values sent to the command port:
;
; 20h - Read Keyboard Controller's Command Byte (KCCB) and send the data to
; the data port (I/O address 60h).
;
; 60h - Write KCCB. The next byte written to I/O address 60h is placed in
; the KCCB. The bits of the KCCB are defined as follows:
;
;       bit 7- Reserved, should be a zero
;       bit 6- IBM industrial computer mode.
;       bit 5- IBM industrial computer mode.
;       bit 4- Disable keyboard.
;       bit 3- Inhibit override.
;       bit 2- System flag
;       bit 1- Reserved, should be a zero.
;       bit 0- Enable output buffer full interrupt.
;
;       AAh - Self test
;       ABh - Interface test
;       ACh - Diagnostic dump
;       ADh - Disable keyboard
;       AEh - Enable keyboard
;       C0h - Read Keyboard Controller input port (equip installed)
;       D0h - Read Keyboard Controller output port
;       D1h - Write Keyboard Controller output port
;       E0h - Read test inputs
;       F0h - FFh - Pulse Output port.
;
; The keyboard controller output port is defined as follows:
;
;       bit 7 - Keyboard data (output)
;       bit 6 - Keyboard clock (output)
;       bit 5 - Input buffer empty
;       bit 4 - Output buffer full
;       bit 3 - undefined
;       bit 2 - undefined
;       bit 1 - Gate A20
;       bit 0 - System reset (0=reset)
;
; The keyboard controller input port is defined as follows:
;
;       bit 7 - Keyboard inhibit switch (0=inhibited)
;       bit 6 - Display switch (0=color, 1= mono)
;       bit 5 - Manufacturing jumper
;       bit 4 - System board RAM (0=disable 2nd 256K RAM on system board).
;       bits 0-3 - undefined.
;
; The keyboard controller status port (64h) is defined as follows:
;
;       bit 1 - Set if input data (60h) not available.
;       bit 0 - Set if output port (60h) cannot accept data.


PutInATBuffer proc      near
              assume    ds:nothing
              pushf
              push      ax
```

```
                push    bx
                push    cx
                push    dx


                mov     dl, al          ;Save char to output.

; Wait until the keyboard controller does not contain data before
; proceeding with shoving stuff down its throat.

                xor     cx, cx
WaitWhlFull:    in      al, 64h
                test    al, 1
                loopnz  WaitWhlFull


; First things first, let's mask the interrupt controller chip (8259) to
; tell it to ignore interrupts coming from the keyboard. However, turn the
; interrupts on so we properly process interrupts from other sources (this
; is especially important because we're going to wind up sending a false
; EOI to the interrupt controller inside the INT 9 BIOS routine).

                cli
                in      al, 21h         ;Get current mask
                push    ax              ;Save intr mask
                or      al, 2           ;Mask keyboard interrupt
                out     21h, al

; Transmit the desired scan code to the keyboard controller. Call this
; byte the new keyboard controller command (we've turned off the keyboard,
; so this won't affect anything).
;
; The following code tells the keyboard controller to take the next byte
; sent to it and use this byte as the KCCB:


                call    WaitToXmit
                mov     al, 60h         ;Write new KCCB command.
                out     64h, al

; Send the scan code as the new KCCB:

                call    WaitToXmit
                mov     al, dl
                out     60h, al

; The following code instructs the system to transmit the KCCB (i.e., the
; scan code) to the system:

                call    WaitToXmit
                mov     al, 20h         ;"Send KCCB" command.
                out     64h, al

                xor     cx, cx
Wait4OutFull:   in      al, 64h
                test    al, 1
                loopz   Wait4OutFull

; Okay, Send a 45h back as the new KCCB to allow the normal keyboard to work
; properly.

                call    WaitToXmit
                mov     al, 60h
                out     64h, al

                call    WaitToXmit
                mov     al, 45h
                out     60h, al

; Okay, execute an INT 9 routine so the BIOS (or whoever) can read the key
; we just stuffed into the keyboard controller. Since we've masked INT 9
; at the interrupt controller, there will be no interrupt coming along from
; the key we shoved in the buffer.
```

```
DoInt9:         in          al, 60h             ;Prevents ints from some codes.
                int         9                   ;Simulate hardware kbd int.


; Just to be safe, reenable the keyboard:

                call        WaitToXmit
                mov         al, 0aeh
                out         64h, al

; Okay, restore the interrupt mask for the keyboard in the 8259a.

                pop         ax
                out         21h, al

                pop         dx
                pop         cx
                pop         bx
                pop         ax
                popf
                ret
PutInATBuffer   endp



; WaitToXmit-   Wait until it's okay to send a command byte to the keyboard
;               controller port.

WaitToXmit      proc        near
                push        cx
                push        ax
                xor         cx, cx
TstCmdPortLp:   in          al, 64h
                test        al, 2           ;Check cntrlr input buffer full flag.
                loopnz      TstCmdPortLp
                pop         ax
                pop         cx
                ret
WaitToXmit      endp



;*****************************************************************************
;
; PutInPS2Buffer- Like PutInATBuffer, it uses the keyboard controller chip
;                 to return the keycode. However, PS/2 compatible controllers
;                 have an actual command to return keycodes.

PutInPS2Buffer  proc        near
                pushf
                push        ax
                push        bx
                push        cx
                push        dx

                mov         dl, al              ;Save char to output.

; Wait until the keyboard controller does not contain data before
; proceeding with shoving stuff down its throat.

                xor         cx, cx
WaitWhlFull:    in          al, 64h
                test        al, 1
                loopnz      WaitWhlFull


; The following code tells the keyboard controller to take the next byte
; sent to it and return it as a scan code.


                call        WaitToXmit
                mov         al, 0d2h            ;Return scan code command.
                out         64h, al
```

```
; Send the scan code:

                call    WaitToXmit
                mov     al, dl
                out     60h, al

                pop     dx
                pop     cx
                pop     bx
                pop     ax
                popf
                ret
PutInPS2Buffer endp


; Main program - Simulates some keystrokes to demo the above code.

Main            proc

                mov     ax, cseg
                mov     ds, ax

                print
                byte    "Simulating keystrokes via Trace Flag",cr,lf
                byte    "This program places 'DIR' in the keyboard buffer"
                byte    cr,lf,0

                mov     al, 20h         ;"D" down scan code
                call    PutInATBuffer
                mov     al, 0a0h        ;"D" up scan code
                call    PutInATBuffer

                mov     al, 17h         ;"I" down scan code
                call    PutInATBuffer
                mov     al, 97h         ;"I" up scan code
                call    PutInATBuffer

                mov     al, 13h         ;"R" down scan code
                call    PutInATBuffer
                mov     al, 93h         ;"R" up scan code
                call    PutInATBuffer

                mov     al, 1Ch         ;Enter down scan code
                call    PutInATBuffer
                mov     al, 9Ch         ;Enter up scan code
                call    PutInATBuffer

                ExitPgm
Main            endp


cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

20.8 Summary

This chapter might seem excessively long for such a mundane topic as keyboard I/O. After all, the Standard Library provides only one primitive routine for keyboard input, getc. However, the keyboard on the PC is a complex beast, having no less than two specialized microprocessors controlling it. These microprocessors accept commands from the PC and send commands and data to the PC. If you want to

write some tricky keyboard handling code, you need to have a firm understanding of the keyboard's underlying hardware.

This chapter began by describing the actions the system takes when a user presses a key. As it turns out, the system transmits two *scan codes* every time you press a key – one scan code when you press the key and one scan code when you release the key. These are called down codes and up codes, accordingly. The scan codes the keyboard transmits to the system have little relationship to the standard ASCII character set. Instead, the keyboard uses its own character set and relies upon the keyboard interrupt service routine to translate these scan codes to their appropriate ASCII codes. Some keys do not have ASCII codes, for these keys the system passes along an *extended key code* to the application requesting keyboard input. While translating scan codes to ASCII codes, the keyboard interrupt service routine makes use of certain BIOS flags that track the position of the *modifier* keys. These keys include the shift, ctrl, alt, capslock, and numlock keys. These keys are known as modifiers because the modify the normal code produced by keys on the keyboard. The keyboard interrupt service routine stuffs incoming characters in the system *type ahead buffer* and updates other BIOS variables in segment 40h. An application program or other system service can access this data prepared by the keyboard interrupt service routine. For more information, see

- "Keyboard Basics" on page 1153

The PC interfaces to the keyboard using two separate microcontroller chips. These chips provide user programming registers and a very flexible command set. If you want to program the keyboard beyond simply reading the keystrokes produced by the keyboard (i.e., manipulate the LEDs on the keyboard), you will need to become familiar with the registers and command sets of these microcontrollers. The discussion of these topics appears in

- "The Keyboard Hardware Interface" on page 1159

Both DOS and BIOS provide facilities to read a key from the system's type ahead buffer. As usual, BIOS' functions provide the most flexibility in terms of getting at the hardware. Furthermore, the BIOS int 16h routine lets you check shift key status, stuff scan/ASCII codes into the type ahead buffer, adjust the autorepeat rate, and more. Given this flexibility, it is difficult to understand why someone would want to talk directly to the keyboard hardware, especially considering the compatibility problems that seem to plague such projects. To learn the proper way to read characters from the keyboard, and more, see

- "The Keyboard DOS Interface" on page 1167
- "The Keyboard BIOS Interface" on page 1168

Although accessing the keyboard hardware directly is a bad idea for most applications, there is a small class of programs, like keyboard enhancers and pop-up programs, that really do need to access the keyboard hardware directly. These programs must supply an interrupt service routine for the int 9 (keyboard) interrupt. For all the details, see:

- "The Keyboard Interrupt Service Routine" on page 1174
- "Patching into the INT 9 Interrupt Service Routine" on page 1184

A keyboard macro program (keyboard enhancer) is a perfect example of a program that might need to talk directly to the keyboard hardware. One problem with such programs is that they need to pass characters along to some underlying application. Given the nature of applications present in the world, this can be a difficult task if you want to be compatible with a large number of PC applications. The problems, and some solutions, appear in

- "Simulating Keystrokes" on page 1186
- "Stuffing Characters in the Type Ahead Buffer" on page 1186
- "Using the 80x86 Trace Flag to Simulate IN AL, 60H Instructions" on page 1187
- "Using the 8042 Microcontroller to Simulate Keystrokes" on page 1192

The PC Parallel Ports Chapter 21

The original IBM PC design provided support for three parallel printer ports that IBM designated LPT1:, LPT2:, and LPT3:[1]. IBM probably envisioned machines that could support a standard dot matrix printer, a daisy wheel printer, and maybe some other auxiliary type of printer for different purposes, all on the same machine (laser printers were still a few years in the future at that time). Surely IBM did not anticipate the general use that parallel ports have received or they would probably have designed them differently. Today, the PC's parallel port controls keyboards, disk drives, tape drives, SCSI adapters, ethernet (and other network) adapters, joystick adapters, auxiliary keypad devices, other miscellaneous devices, and, oh yes, printers. This chapter will not attempt to describe how to use the parallel port for all these various purposes – this book is long enough already. However, a thorough discussion of how the parallel interface controls a printer and one other application of the parallel port (cross machine communication) should provide you with enough ideas to implement the next great parallel device.

21.1 Basic Parallel Port Information

There are two basic data transmission methods modern computes employ: parallel data transmission and serial data transmission. In a serial data transmission scheme (see "The PC Serial Ports" on page 1223) one device sends data to another a single bit at a time across one wire. In a parallel transmission scheme, one device sends data to another several bits at a time (in parallel) on several different wires. For example, the PC's parallel port provides eight data lines compared to the serial port's single data line. Therefore, it would seem that the parallel port would be able to transmit data eight times as fast since there are eight times as many wires in the cable. Likewise, it would seem that a serial cable, for the same price as a parallel cable, would be able to go eight times as far since there are fewer wires in the cable. And these are the common trade-offs typically given for parallel vs. serial communication methods: speed vs. cost.

In practice, parallel communications is not eight times faster than serial communications, nor do parallel cables cost eight times as much. In generally, those who design serial cables (.e.g, ethernet cables) use higher materials and shielding. This raises the cost of the cable, but allows devices to transmit data, still a bit at a time, much faster. Furthermore, the better cable design allows greater distances between devices. Parallel cables, on the other hand, are generally quite inexpensive and designed for very short connections (generally no more than about six to ten feet). The real world problems of electrical noise and cross-talk create problems when using long parallel cables and limit how fast the system can transmit data. In fact the original Centronics printer port specification called for no more than 1,000 characters/second data transmission rate, so many printers were designed to handle data at this transmission rate. Most parallel ports can easily outperform this value; however, the limiting factor is still the cable, not any intrinsic limitation in a modern computer.

Although a parallel communication system could use any number of wires to transmit data, most parallel systems use eight data lines to transmit a byte at a time. There are a few notable exceptions. For example, the SCSI interface is a parallel interface, yet newer versions of the SCSI standard allow eight, sixteen, and even thirty-two bit data transfers. In this chapter we will concentrate on byte-sized transfers since the parallel port on the PC provides for eight-bit data.

A typical parallel communication system can be one way (or *unidirectional*) or two way (*bidirectional*). The PC's parallel port generally supports unidirectional communications (from the PC to the printer), so we will consider this simpler case first.

In a unidirectional parallel communication system there are two distinguished sites: the transmitting site and the receiving site. The transmitting site places its data on the data lines and informs the receiving site that data is available; the receiving site then reads the data lines and informs the transmitting site that it

1. In theory, the BIOS allows for a fourth parallel printer port, LPT4:, but few (if any) adapter cards have ever been built that claim to work as LPT4:.

has taken the data. Note how the two sites synchronize their access to the data lines – the receiving site does not read the data lines until the transmitting site tells it to, the transmitting site does not place a new value on the data lines until the receiving site removes the data and tells the transmitting site that it has the data. *Handshaking* is the term that describes how these two sites coordinate the data transfer.

To properly implement handshaking requires two additional lines. The *strobe* (or data strobe) line is what the transmitting site uses to tell the receiving site that data is available. The *acknowledge* line is what the receiving site uses to tell the transmitting site that it has taken the data and is ready for more. The PC's parallel port actually provides a third handshaking line, *busy*, that the receiving site can use to tell the transmitting site that it is busy and the transmitting site should not attempt to send data. A typical data transmission session looks something like the following:

Transmitting site:

1) The transmitting site checks the busy line to see if the receiving is busy. If the busy line is active, the transmitter waits in a loop until the busy line becomes inactive.
2) The transmitting site places its data on the data lines.
3) The transmitting site activates the strobe line.
4) The transmitting site waits in a loop for the acknowledge line to become active.
5) The transmitting site sets the strobe inactive.
6) The transmitting site waits in a loop for the acknowledge line to become inactive.
7) The transmitting site repeats steps one through six for each byte it must transmit.

Receiving site:

1) The receiving site sets the busy line inactive (assuming it is ready to accept data).
2) The receiving site waits in a loop until the strobe line becomes active.
3) The receiving site reads the data from the data lines (and processes the data, if necessary).
4) The receiving site activates the acknowledge line.
5) The receiving site waits in a loop until the strobe line goes inactive.
6) The receiving site sets the acknowledge line inactive.
7) The receiving site repeats steps one through six for each additional byte it must receive.

By carefully following these steps, the receiving and transmitting sites carefully coordinate their actions so the transmitting site doesn't attempt to put several bytes on the data lines before the receiving site consumes them and the receiving site doesn't attempt to read data that the transmitting site has not sent.

Bidirectional data transmission is often nothing more than two unidirectional data transfers with the roles of the transmitting and receiving sites reversed for the second communication channel. Some PC parallel ports (particularly on PS/2 systems and many notebooks) provide a bidirectional parallel port. Bidirectional data transmission on such hardware is slightly more complex than on systems that implement bidirectional communication with two unidirectional ports. Bidirectional communication on a bidirectional parallel port requires an extra set of control lines so the two sites can determine who is writing to the common data lines at any one time.

21.2 The Parallel Port Hardware

The standard unidirectional parallel port on the PC provides more than the 11 lines described in the previous section (eight data, three handshake). The PC's parallel port provides the following signals:

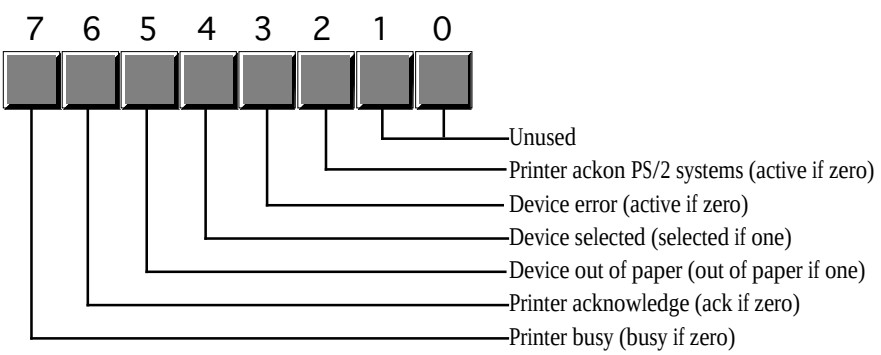
Table 79: Parallel Port Signals

Pin Number on Connector	I/O Direction	Active Polarity	Signal Description
1	output	0	Strobe (data available signal).
2-9	output	-	Data lines (bit 0 is pin 2, bit 7 is pin 9).
10	input	0	Acknowledge line (active when remote system has taken data).
11	input	0	Busy line (when active, remote system is busy and cannot accept data).
12	input	1	Out of paper (when active, printer is out of paper).
13	input	1	Select. When active, the printer is selected.
14	output	0	Autofeed. When active, the printer automatically inserts a line feed after every carriage return it receives.
15	input	0	Error. When active, there is a printer error.
16	output	0	Init. When held active for at least 50 µsec, this signal causes the printer to initialize itself.
17	output	0	Select input. This signal, when inactive, forces the printer off-line
18-25	-	-	Signal ground.

Note that the parallel port provides 12 output lines (eight data lines, strobe, autofeed, init, and select input) and five input lines (acknowledge, busy, out of paper, select, and error). Even though the port is unidirectional, there is a good mixture of input and output lines available on the port. Many devices (like disk and tape drives) that require bidirectional data transfer use these extra lines to perform bidirectional data transfer.

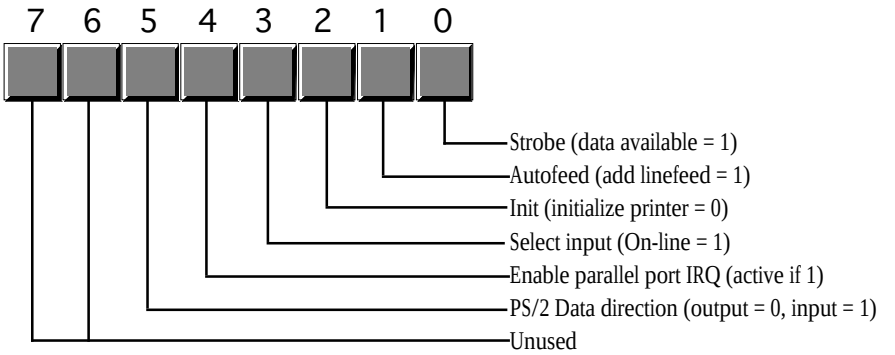
On bidirectional parallel ports (found on PS/2 and laptop systems), the strobe and data lines are both input and output lines. There is a bit in a control register associated with the parallel port that selects the transfer direction at any one given instant (you cannot transfer data in both direction simultaneously).

There are three I/O addresses associated with a typical PC compatible parallel port. These addresses belong to the *data register*, *the status register*, and *the control register*. The data register is an eight-bit read/write port. Reading the data register (in a unidirectional mode) returns the value last written to the data register. The control and status registers provide the interface to the other I/O lines. The organization of these ports is as follows:



Parallel Port Status Register (read only)

Bit two (printer acknowledge) is available only on PS/2 and other systems that support a bidirectional printer port. Other systems do not use this bit.



Parallel Port Control Register

The parallel port control register is an output register. Reading this location returns the last value written to the control register *except for bit five* that is write only. Bit five, the data direction bit, is available only on PS/2 and other systems that support a bidirectional parallel port. If you write a zero to this bit, the strobe and data lines are output bits, just like on the unidirectional parallel port. If you write a one to this bit, then the data and strobe lines are inputs. Note that in the input mode (bit 5 = 1), bit zero of the control register is actually an input. Note: writing a one to bit four of the control register enables the printer IRQ (IRQ 7). However, this feature does not work on all systems so very few programs attempt to use interrupts with the parallel port. When active, the parallel port will generate an int 0Fh whenever the printer acknowledges a data transmission.

Since the PC supports up to three separate parallel ports, there could be as many as three sets of these parallel port registers in the system at any one time. There are three *parallel port base addresses* associated with the three possible parallel ports: 3BCh, 378h, and 278h. We will refer to these as the base addresses for LPT1:, LPT2:, and LPT3:, respectively. The parallel port data register is always located at the base address for a parallel port, the status register appears at the base address plus one, and the control register appears at the base address plus two. For example, for LPT1:, the data register is at I/O address 3BCh, the status register is at I/O address 3BDh, and the control register is at I/O address 3BEh.

There is one minor glitch. The I/O addresses for LPT1:, LPT2:, and LPT3: given above are the *physical addresses* for the parallel ports. The BIOS provides *logical addresses* for these parallel ports as well. This lets users remap their printers (since most software only writes to LPT1:). To accomplish this, the BIOS reserves eight bytes in the BIOS variable space (40:8, 40:0A, 40:0C, and 40:0E). Location 40:8 contains the base address for logical LPT1:, location 40:0A contains the base address for logical LPT2:, etc. When software accesses LPT1:, LPT2:, etc., it generally accesses the parallel port whose base address appears in one of these locations.

21.3 Controlling a Printer Through the Parallel Port

Although there are many devices that connect to the PC's parallel port, printers still make up the vast number of such connections. Therefore, describing how to control a printer from the PC's parallel port is probably the best first example to present. As with the keyboard, your software can operate at three different levels: it can print data using DOS, using BIOS, or by writing directly to the parallel port hardware. As with the keyboard interface, using DOS or BIOS is the best approach if you want to maintain compatibility with other devices that plug into the parallel port[2]. Of course, if you are controlling some other type of

2. Many devices connect to the parallel port with a pass-through plug allowing you to use that device and still use the parallel port for your printer. However, if you talk directly to the parallel port with your software, it may conflict with that device's operation.

device, going directly to the hardware is your only choice. However, the BIOS provides good printer support, so going directly to the hardware is rarely necessary if you simply want to send data to the printer.

21.3.1 Printing via DOS

MS-DOS provides two calls you can use to send data to the printer. DOS function 05h writes the character in the dl register directly to the printer. Function 40h, with a file handle of 04h, also sends data to the printer. Since the chapter on DOS and BIOS fully describes these functions, we will not discuss them any further here. For more information, see “MS-DOS, PC-BIOS, and File I/O” on page 699 .

21.3.2 Printing via BIOS

Although DOS provides a reasonable set of functions to send characters to the printer, it does not provide functions to let you initialize the printer or obtain the current printer status. Furthermore, DOS only prints to LPT1:. The PC's int 17h BIOS routine provides three functions, print, initialize, and status. You can apply these functions to any supported parallel port on the system. The print function is roughly equivalent to DOS' print character function. The initialize function initializes the printer using system dependent timing information. The printer status returns the information from the printer status port along with time-out information. For more information on these routines, see “MS-DOS, PC-BIOS, and File I/O” on page 699.

21.3.3 An INT 17h Interrupt Service Routine

Perhaps the best way to see how the BIOS functions operate is to write a replacement int 17h ISR for a printer. This section explains the handshaking protocol and variables the printer driver uses. It also describes the operation and return results associated with each machine.

There are eight variables in the BIOS variable space (segment 40h) the printer driver uses. The following table describes each of these variables:

Table 80: BIOS Parallel Port Variables

Address	Description
40:08	Base address of LPT1: device.
40:0A	Base address of LPT2: device.
40:0C	Base address of LPT3: device.
40:0E	Base address of LPT4: device.
40:78	LPT1: time-out value. The printer port driver software should return an error if the printer device does not respond in a reasonable amount of time. This variable (if non-zero) determines how many loops of 65,536 iterations each a driver will wait for a printer acknowledge. If zero, the driver will wait forever.
40:79	LPT2: time-out value. See description above.
40:7A	LPT3: time-out value. See description above.
40:7B	LPT4: time-out value. See description above.

You will notice a slight deviation in the handshake protocol in the following code. This printer driver does not wait for an acknowledge from the printer *after* sending a character. Instead, it checks to see if

the printer has sent an acknowledge to the previous character *before* sending a character. This saves a small amount of time because the program printer then characters can continue to operating in parallel with the receipt of the acknowledge from the printer. You will also notice that this particular driver does not monitor the busy lines. Almost every printer in existence leaves this line inactive (not busy), so there is no need to check it. If you encounter a printer than does manipulate the busy line, the modification to this code is trivial. The following code implements the int 17h service:

```
; INT17.ASM
;
; A short passive TSR that replaces the BIOS' int 17h handler.
; This routine demonstrates the function of each of the int 17h
; functions that a standard BIOS would provide.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg            segment    para public 'code'
cseg            ends

; Marker segment, to find the end of the resident section.

EndResident     segment    para public 'Resident'
EndResident     ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


byp             equ        <byte ptr>

cseg            segment    para public 'code'
                assume     cs:cseg, ds:cseg

OldInt17        dword      ?


; BIOS variables:

PrtrBase        equ        8
PrtrTimeOut     equ        78h



; This code handles the INT 17H operation. INT 17H is the BIOS routine
; to send data to the printer and report on the printer's status. There
; are three different calls to this routine, depending on the contents
; of the AH register. The DX register contains the printer port number.
;
; DX=0 -- Use LPT1:
; DX=1 -- Use LPT2:
; DX=2 -- Use LPT3:
; DX=3 -- Use LPT4:
;
; AH=0 --       Print the character in AL to the printer. Printer status is
;               returned in AH. If bit #0 = 1 then a timeout error occurred.
;
; AH=1 --       Initialize printer. Status is returned in AH.
;
; AH=2 --       Return printer status in AH.
;
;
; The status bits returned in AH are as follows:
;
```

```
; Bit    Function                      Non-error values
; ---    -------------------------     ----------------
;  0     1=time out error                 0
;  1     unused                           x
;  2     unused                           x
;  3     1=I/O error                      0
;  4     1=selected, 0=deselected.        1
;  5     1=out of paper                   0
;  6     1=acknowledge                    x
;  7     1=not busy                       x
;
; Note that the hardware returns bit 3 with zero if an error has occurred,
; with one if there is no error. The software normally inverts this bit
; before returning it to the caller.
;
;
; Printer port hardware locations:
;
; There are three ports used by the printer hardware:
;
; PrtrPortAdrs ---       Output port where data is sent to printer (8 bits).
; PrtrPortAdrs+1 ---     Input port where printer status can be read (8 bits).
; PrtrPortAdrs+2 ---     Output port where control information is sent to the
;                        printer.
;
; Data output port- 8-bit data is transmitted to the printer via this port.
;
; Input status port:
;             bit 0:     unused.
;             bit 1:     unused.
;             bit 2:     unused.
;
;             bit 3:     -Error, normally this bit means that the
;                        printer has encountered an error. However,
;                        with the P101 installed this is a data
;                        return line for the keyboard scan.
;
;             bit 4:     +SLCT, normally this bit is used to determine
;                        if the printer is selected or not. With the
;                        P101 installed this is a data return
;                        line for the keyboard scan.
;
;             bit 5:     +PE, a 1 in this bit location means that the
;                        printer has detected the end of paper. On
;                        many printer ports, this bit has been found
;                        to be inoperative.
;
;             bit 6:     -ACK, A zero in this bit position means that
;                        the printer has accepted the last character
;                        and is ready to accept another. This bit
;                        is not normally used by the BIOS as bit 7
;                        also provides this function (and more).
;
;             bit 7:     -Busy, When this signal is active (0) the
;                        printer is busy and cannot accept data.
;                        When this bit is set to one, the printer
;                        can accept another character.
;
;
;
; Output control port:
;
;             Bit 0:     +Strobe, A 0.5 us (minimum) active high pulse
;                        on this bit clocks the data latched into the
;                        printer data output port to the printer.
;
;             Bit 1:     +Auto FD XT - A 1 stored at this bit causes
;                        the printer to line feed after a line is
;                        printed. On some printer interfaces (e.g.,
;                        the Hercules Graphics Card) this bit is
;                        inoperative.
;
;             Bit 2:     -INIT, a zero on this bit (for a minimum of
;                        50 us) will cause the printer to (re)init-
```

```
;                       ialize itself.
;
;               Bit 3:  +SLCT IN, a one in this bit selects the
;                       printer. A zero will cause the printer to
;                       go off-line.
;
;               Bit 4:  +IRQ ENABLE, a one in this bit position
;                       allows an interrupt to occur when -ACK
;                       changes from one to zero.
;
;               Bit 5:  Direction control on BI-DIR port. 0=output,
;                       1=input.
;               Bit 6:  reserved, must be zero.
;               Bit 7:  reserved, must be zero.

MyInt17         proc    far
                assume  ds:nothing

                push    ds
                push    bx
                push    cx
                push    dx

                mov     bx, 40h         ;Point DS at BIOS vars.
                mov     ds, bx

                cmp     dx, 3           ;Must be LPT1..LPT4.
                ja      InvalidPrtr

                cmp     ah, 0           ;Branch to the appropriate code for
                jz      PrtChar         ; the printer function
                cmp     ah, 2
                jb      PrtrInit
                je      PrtrStatus

; If they passed us an opcode we don't know about, just return.

InvalidPrtr:    jmp     ISR17Done



; Initialize the printer by pulsing the init line for at least 50 us.
; The delay loop below will delay well beyond 50 usec even on the fastest
; machines.

PrtrInit:       mov     bx, dx          ;Get printer port value.
                shl     bx, 1           ;Convert to byte index.
                mov     dx, PrtrBase[bx]  ;Get printer base address.
                test    dx, dx          ;Does this printer exist?
                je      InvalidPrtr     ;Quit if no such printer.
                add     dx, 2           ;Point dx at control reg.
                in      al, dx          ;Read current status.
                and     al, 11011011b   ;Clear INIT/BIDIR bits.
                out     dx, al          ;Reset printer.
                mov     cx, 0           ;This will produce at least
PIDelay:        loop    PIDelay         ; a 50 usec delay.
                or      al, 100b        ;Stop resetting printer.
                out     dx, al
                jmp     ISR17Done


; Return the current printer status. This code reads the printer status
; port and formats the bits for return to the calling code.

PrtrStatus:     mov     bx, dx          ;Get printer port value.
                shl     bx, 1           ;Convert to byte index.
                mov     dx, PrtrBase[bx]  ;Base address of printer port.
                mov     al, 00101001b   ;Dflt: every possible error.
                test    dx, dx          ;Does this printer exist?
                je      InvalidPrtr     ;Quit if no such printer.
                inc     dx              ;Point at status port.
                in      al, dx          ;Read status port.
                and     al, 11111000b   ;Clear unused/timeout bits.
                jmp     ISR17Done
```

```
; Print the character in the accumulator!

PrtChar:      mov       bx, dx
              mov       cl, PrtrTimeOut[bx] ;Get time out value.
              shl       bx, 1               ;Convert to byte index.
              mov       dx, PrtrBase[bx]  ;Get Printer port address
              or        dx, dx              ;Non-nil pointer?
              jz        NoPrtr2             ; Branch if a nil ptr

; The following code checks to see if an acknowlege was received from
; the printer. If this code waits too long, a time-out error is returned.
; Acknowlege is supplied in bit #7 of the printer status port (which is
; the next address after the printer data port).

              push      ax
              inc       dx                  ;Point at status port
              mov       bl, cl              ;Put timeout value in bl
              mov       bh, cl              ; and bh.
WaitLp1:      xor       cx, cx              ;Init count to 65536.
WaitLp2:      in        al, dx              ;Read status port
              mov       ah, al              ;Save status for now.
              test      al, 80h             ;Printer acknowledge?
              jnz       GotAck              ;Branch if acknowledge.
              loop      WaitLp2             ;Repeat 65536 times.
              dec       bl                  ;Decrement time out value.
              jnz       WaitLp1             ;Repeat 65536*TimeOut times.

; See if the user has selected no timeout:

              cmp       bh, 0
              je        WaitLp1

; TIMEOUT ERROR HAS OCCURRED!
;
; A timeout - I/O error is returned to the system at this point.
; Either we fall through to this point from above (time out error) or
; the referenced printer port doesn't exist. In any case, return an error.

NoPrtr2:      or        ah, 9               ;Set timeout-I/O error flags
              and       ah, 0F9h            ;Turn off unused flags.
              xor       ah, 40h             ;Flip busy bit.

; Okay, restore registers and return to caller.

              pop       cx                  ;Remove old ax.
              mov       al, cl              ;Restore old al.
              jmp       ISR17Done


; If the printer port exists and we've received an acknowlege, then it's
; okay to transmit data to the printer. That job is handled down here.

GotAck:       mov       cx, 16              ;Short delay if crazy prtr
GALp:         loop      GALp                ; needs hold time after ack.
              pop       ax                  ;Get char to output and
              push      ax                  ; save again.
              dec       dx                  ;Point DX at printer port.
              pushf                         ;Turn off interrupts for now.
              cli
              out       dx, al              ;Output data to the printer.

; The following short delay gives the data time to travel through the
; parallel lines. This makes sure the data arrives at the printer before
; the strobe (the times can vary depending upon the capacitance of the
; parallel cable's lines).

              mov       cx, 16              ;Give data time to settle
DataSettleLp: loop      DataSettleLp        ; before sending strobe.

; Now that the data has been latched on the printer data output port, a
; strobe must be sent to the printer. The strobe line is connected to
```

```
; bit zero of the control port. Also note that this clears bit 5 of the
; control port. This ensures that the port continues to operate as an
; output port if it is a bidirectional device. This code also clears bits
; six and seven which IBM claims should be left zero.

                inc     dx              ;Point DX at the printer
                inc     dx              ; control output port.
                in      al, dx          ;Get current control bits.
                and     al, 01eh        ;Force strobe line to zero and
                out     dx, al          ; make sure it's an output port.


                mov     cx, 16          ;Short delay to allow data
Delay0:         loop    Delay0          ; to become good.

                or      al, 1           ;Send out the (+) strobe.
                out     dx, al          ;Output (+) strobe to bit 0

                mov     cx, 16          ;Short delay to lengthen strobe
StrobeDelay:    loop    StrobeDelay

                and     al, 0FEh        ;Clear the strobe bit.
                out     dx, al          ;Output to control port.
                popf                    ;Restore interrupts.

                pop     dx              ;Get old AX value
                mov     al, dl          ;Restore old AL value

ISR17Done:      pop     dx
                pop     cx
                pop     bx
                pop     ds
                iret
MyInt17         endp


Main            proc

                mov     ax, cseg
                mov     ds, ax

                print
                byte    "INT 17h Replacement",cr,lf
                byte    "Installing....",cr,lf,0

; Patch into the INT 17 interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 17 value directly into
; the OldInt17 variable.

                cli                     ;Turn off interrupts!
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[17h*4]
                mov     word ptr OldInt17, ax
                mov     ax, es:[17h*4 + 2]
                mov     word ptr OldInt17+2, ax
                mov     es:[17h*4], offset MyInt17
                mov     es:[17h*4+2], cs
                sti                     ;Okay, ints back on.


; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte    "Installed.",cr,lf,0

                mov     ah, 62h         ;Get this program's PSP
                int     21h             ; value.

                mov     dx, EndResident;Compute size of program.
                sub     dx, bx
                mov     ax, 3100h       ;DOS TSR command.
```

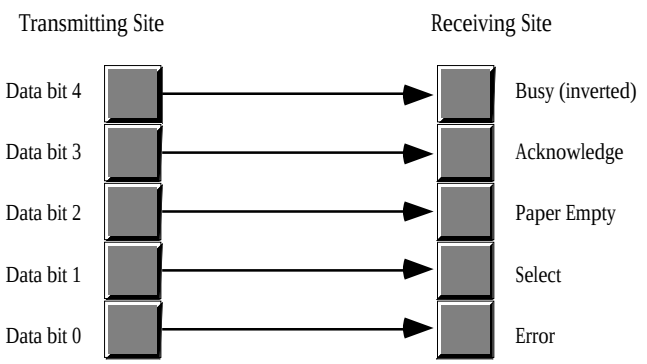
```
                int         21h
Main            endp
cseg            ends

sseg            segment     para stack 'stack'
stk             byte        1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

21.4 Inter-Computer Communications on the Parallel Port

Although printing is, by far, the most popular use for the parallel port on a PC, many devices use the parallel port for other purposes, as mentioned earlier. It would not be fitting to close this chapter without at least one example of a non-printer application for the parallel port. This section will describe how to get two computers to transmit files from one to the other across the parallel port.

The Laplink™program from Travelling Software is a good example of a commercial product that can transfer data across the PC's parallel port; although the following software is not as robust or feature laden as Laplink, it does demonstrate the basic principles behind such software.

Note that you cannot connect two computer's parallel ports with a simple cable that has DB25 connectors at each end. In fact, doing so could damage the computers' parallel ports because you'd be connecting digital outputs to digital outputs (a real no-no). However, you purchase "Laplink compatible" cables (or buy *real* Laplink cables for that matter) the provide proper connections between the parallel ports of two computers. As you may recall from the section on the parallel port hardware, the unidirectional parallel port provides five input signals. A Laplink cable routes four of the data lines to four of these input lines in both directions. The connections on a Laplink compatible cable are as follows:



Connections on a Laplink Compatible Cable

Data written on bits zero through three of the data register at the transmitting site appear, unchanged, on bits three through six of the status port on the receiving site. Bit four of the transmitting site appears, inverted, at bit seven of the receiving site. Note that Laplink compatible cables are bidirectional. That is, you can transmit data from either site to the other using the connections above. However, since there are only five input bits on the parallel port, you must transfer the data four bits at a time (we need one bit for the data strobe). Since the receiving site needs to acknowledge data transmissions, we cannot simultaneously transmit data in both directions. We must use one of the output lines at the site receiving data to acknowledge the incoming data.

Since the two sites cooperating in a data transfer across the parallel cable must take turns transmitting and receiving data, we must develop a *protocol* so each participant in the data transfer knows when it is okay to transmit and receive. Our protocol will be very simple – a site is either a transmitter or a receiver, the roles will never switch. Designing a more complex protocol is not difficult, but this simple protocol will suffice for the example you are about to see. Later in this section we will discuss ways to develop a protocol that allows two-way transmissions.

The following example programs will transmit and receive a single file across the parallel port. To use this software, you run the *transmit* program on the transmitting site and the *receive* program on the receiving site. The transmission program fetches a file name from the DOS command line and opens that file for reading (generating an error, and quitting, if the file does not exist). Assuming the file exists, the transmit program then queries the receiving site to see if it is available. The transmitter checks for the presence of the receiving site by alternately writing zeros and ones to all output bits then reading its input bits. The receiving site will invert these values and write them back when it comes on-line. Note that the order of execution (transmitter first or receiver first) does not matter. The two programs will attempt to handshake until the other comes on line.When both sites cycle through the inverting values three times, they write the value 05h to their output ports to tell the other site they are ready to proceed. A time-out function aborts either program if the other site does not respond in a reasonable amount of time.

Once the two sites are synchronized, the transmitting site determines the size of the file and then transmits the file name and size to the receiving site. The receiving site then begins waiting for the receipt of data.

The transmitting site sends the data 512 bytes at a time to the receiving site. After the transmission of 512 bytes, the receiving site delays sending an acknowledgment and writes the 512 bytes of data to the disk. Then the receiving site sends the acknowledge and the transmitting site begins sending the next 512 bytes. This process repeats until the receiving site has accepted all the bytes from the file.

Here is the code for the transmitter:

```
; TRANSMIT.ASM
;
; This program is the transmitter portion of the programs that transmit files
; across a Laplink compatible parallel cable.
;
; This program assumes that the user want to use LPT1: for transmission.
; Adjust the equates, or read the port from the command line if this
; is inappropriate.

                .286
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment    para public 'data'

TimeOutConst    equ        4000              ;About 1 min on 66Mhz 486.
PrtrBase        equ        10                ;Offset to LPT1: adrs.

MyPortAdrs      word       ?                 ;Holds printer port address.
FileHandle      word       ?                 ;Handle for output file.
FileBuffer      byte       512 dup (?)    ;Buffer for incoming data.

FileSize dword ?                          ;Size of incoming file.
FileNamePtr     dword      ?                 ;Holds ptr to filename

dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg


; TestAbort-    Check to see if the user has pressed ctrl-C and wants to
;               abort this program. This routine calls BIOS to see if the
```

```
;               user has pressed a key. If so, it calls DOS to read the
;               key (function AH=8, read a key w/o echo and with ctrl-C
;               checking).

TestAbort       proc        near
                push        ax
                push        cx
                push        dx
                mov         ah, 1
                int         16h                 ;See if keypress.
                je          NoKeyPress          ;Return if no keypress.
                mov         ah, 8               ;Read char, chk for ctrl-C.
                int         21h                 ;DOS aborts if ctrl-C.
NoKeyPress:     pop         dx
                pop         cx
                pop         ax
                ret
TestAbort       endp



; SendByte-     Transmit the byte in AL to the receiving site four bits
;               at a time.

SendByte        proc        near
                push        cx
                push        dx
                mov         ah, al              ;Save byte to xmit.

                mov         dx, MyPortAdrs      ;Base address of LPT1: port.

; First, just to be sure, write a zero to bit #4. This reads as a one
; in the busy bit of the receiver.

                mov         al, 0
                out         dx, al              ;Data not ready yet.

; Wait until the receiver is not busy. The receiver will write a zero
; to bit #4 of its data register while it is busy. This comes out as a
; one in our busy bit (bit 7 of the status register). This loop waits
; until the receiver tells us its ready to receive data by writing a
; one to bit #4 (which we read as a zero). Note that we check for a
; ctrl-C every so often in the event the user wants to abort the
; transmission.

                inc         dx                  ;Point at status register.
W4NBLp:         mov         cx, 10000
Wait4NotBusy:   in          al, dx              ;Read status register value.
                test        al, 80h             ;Bit 7 = 1 if busy.
                loopne      Wait4NotBusy        ;Repeat while busy, 10000 times.
                je          ItsNotbusy          ;Leave loop if not busy.
                call        TestAbort           ;Check for Ctrl-C.
                jmp         W4NBLp

; Okay, put the data on the data lines:

ItsNotBusy:     dec         dx                  ;Point at data register.
                mov         al, ah              ;Get a copy of the data.
                and         al, 0Fh             ;Strip out H.O. nibble
                out         dx, al              ;"Prime" data lines, data not avail.
                or          al, 10h             ;Turn data available on.
                out         dx, al              ;Send data w/data available strobe.

; Wait for the acknowledge from the receiving site. Every now and then
; check for a ctrl-C so the user can abort the transmission program from
; within this loop.

                inc         dx              ;Point at status register.
W4ALp:          mov         cx, 10000       ;Times to loop between ctrl-C checks.
Wait4Ack:       in          al, dx          ;Read status port.
                test        al, 80h         ;Ack = 1 when rcvr acknowledges.
                loope       Wait4Ack        ;Repeat 10000 times or until ack.
                jne         GotAck          ;Branch if we got an ack.
                call        TestAbort       ;Every 10000 calls, check for a
```

```
                jmp         W4ALp           ; ctrl-C from the user.

; Send the data not available signal to the receiver:

GotAck:         dec         dx              ;Point at data register.
                mov         al, 0           ;Write a zero to bit 4, this appears
                out         dx, al          ; as a one in the rcvr's busy bit.


; Okay, on to the H.O. nibble:

                inc         dx              ;Point at status register.
W4NB2:          mov         cx, 10000       ;10000 calls between ctrl-C checks.
Wait4NotBsy2:   in          al, dx          ;Read status register.
                test        al, 80h         ;Bit 7 = 1 if busy.
                loopne      Wait4NotBsy2    ;Loop 10000 times while busy.
                je          NotBusy2        ;H.O. bit clear (not busy)?
                call        TestAbort       ;Check for ctrl-C.
                jmp         W4NB2

; Okay, put the data on the data lines:

NotBusy2:       dec         dx              ;Point at data register.
                mov         al, ah          ;Retrieve data to get H.O. nibble.
                shr         al, 4           ;Move H.O. nibble to L.O. nibble.
                out         dx, al          ;"Prime" data lines.
                or          al, 10h         ;Data + data available strobe.
                out         dx, al          ;Send data w/data available strobe.

; Wait for the acknowledge from the receiving site:

                inc         dx              ;Point at status register.
W4A2Lp:         mov         cx, 10000
Wait4Ack2:      in          al, dx          ;Read status port.
                test        al, 80h         ;Ack = 1
                loope       Wait4Ack2       ;While while no acknowledge
                jne         GotAck2         ;H.O. bit = 1 (ack)?
                call        TestAbort       ;Check for ctrl-C
                jmp         W4A2Lp

; Send the data not available signal to the receiver:

GotAck2:        dec         dx              ;Point at data register.
                mov         al, 0           ;Output a zero to bit #4 (that
                out         dx, al          ; becomes busy=1 at rcvr).

                mov         al, ah          ;Restore original data in AL.
                pop         dx
                pop         cx
                ret
SendByte        endp


; Synchronization routines:
;
; Send0s-       Transmits a zero to the receiver site and then waits to
;               see if it gets a set of ones back. Returns carry set if
;               this works, returns carry clear if we do not get a set of
;               ones back in a reasonable amount of time.

Send0s          proc        near
                push        cx
                push        dx

                mov         dx, MyPortAdrs

                mov         al, 0               ;Write the initial zero
                out         dx, al              ; value to our output port.

                xor         cx, cx              ;Checks for ones 10000 times.
Wait41s:        inc         dx                  ;Point at status port.
                in          al, dx              ;Read status port.
                dec         dx                  ;Point back at data port.
```

```
                and     al, 78h         ;Mask input bits.
                cmp     al, 78h         ;All ones yet?
                loopne  Wait41s
                je      Got1s           ;Branch if success.
                clc                     ;Return failure.
                pop     dx
                pop     cx
                ret

Got1s:          stc                     ;Return success.
                pop     dx
                pop     cx
                ret
Send0s          endp


; Send1s-       Transmits all ones to the receiver site and then waits to
;               see if it gets a set of zeros back. Returns carry set if
;               this works, returns carry clear if we do not get a set of
;               zeros back in a reasonable amount of time.

Send1s          proc    near
                push    cx
                push    dx

                mov     dx, MyPortAdrs  ;LPT1: base address.

                mov     al, 0Fh         ;Write the "all ones"
                out     dx, al          ; value to our output port.

                mov     cx, 0
Wait40s:        inc     dx              ;Point at input port.
                in      al, dx          ;Read the status port.
                dec     dx              ;Point back at data port.
                and     al, 78h         ;Mask input bits.
                loopne  Wait40s         ;Loop until we get zero back.
                je      Got0s           ;All zeros? If so, branch.
                clc                     ;Return failure.
                pop     dx
                pop     cx
                ret


Got0s:          stc                     ;Return success.
                pop     dx
                pop     cx
                ret
Send1s          endp



; Synchronize-  This procedure slowly writes all zeros and all ones to its
;               output port and checks the input status port to see if the
;               receiver site has synchronized. When the receiver site
;               is synchronized, it will write the value 05h to its output
;               port. So when this site sees the value 05h on its input
;               port, both sites are synchronized. Returns with the
;               carry flag set if this operation is successful, clear if
;               unsuccessful.

Synchronize     proc    near
                print
                byte    "Synchronizing with receiver program"
                byte    cr,lf,0

                mov     dx, MyPortAdrs

                mov     cx, TimeOutConst  ;Time out delay.
SyncLoop:       call    Send0s            ;Send zero bits, wait for
                jc      Got1s             ; ones (carry set=got ones).

; If we didn't get what we wanted, write some ones at this point and see
; if we're out of phase with the receiving site.
```

```
Retry0:       call      Send1s            ;Send ones, wait for zeros.
              jc        SyncLoop          ;Carry set = got zeros.

; Well, we didn't get any response yet, see if the user has pressed ctrl-C
; to abort this program.

DoRetry:      call      TestAbort

; Okay, the receiving site has yet to respond. Go back and try this again.

              loop      SyncLoop

; If we've timed out, print an error message and return with the carry
; flag clear (to denote a timeout error).

              print
              byte      "Transmit: Timeout error waiting for receiver"
              byte      cr,lf,0
              clc
              ret

; Okay, we wrote some zeros and we got some ones. Let's write some ones
; and see if we get some zeros. If not, retry the loop.

Got1s:
              call      Send1s            ;Send one bits, wait for
              jnc       DoRetry           ; zeros (carry set=got zeros).

; Well, we seem to be synchronized. Just to be sure, let's play this out
; one more time.

              call      Send0s            ;Send zeros, wait for ones.
              jnc       Retry0
              call      Send1s            ;Send ones, wait for zeros.
              jnc       DoRetry

; We're syncronized. Let's send out the 05h value to the receiving
; site to let it know everything is cool:

              mov       al, 05h           ;Send signal to receiver to
              out       dx, al            ; tell it we're sync'd.

              xor       cx, cx            ;Long delay to give the rcvr
FinalDelay:   loop      FinalDelay    ; time to prepare.

              print
              byte      "Synchronized with receiving site"
              byte      cr,lf,0
              stc
              ret
Synchronize   endp



; File I/O routines:
;
; GetFileInfo- Opens the user specified file and passes along the file
;              name and file size to the receiving site. Returns the
;              carry flag set if this operation is successful, clear if
;              unsuccessful.

GetFileInfo   proc      near

; Get the filename from the DOS command line:

              mov       ax, 1
              argv
              mov       word ptr FileNamePtr, di
              mov       word ptr FileNamePtr+2, es

              printf
              byte      "Opening %^s\n",0
              dword     FileNamePtr
```

```
; Open the file:

                push    ds
                mov     ax, 3D00h       ;Open for reading.
                lds     dx, FileNamePtr
                int     21h
                pop     ds
                jc      BadFile
                mov     FileHandle, ax

; Compute the size of the file (do this by seeking to the last position
; in the file and using the return position as the file length):

                mov     bx, ax          ;Need handle in BX.
                mov     ax, 4202h       ;Seek to end of file.
                xor     cx, cx          ;Seek to position zero
                xor     dx, dx          ; from the end of file.
                int     21h
                jc      BadFile

; Save final position as file length:

                mov     word ptr FileSize, ax
                mov     word ptr FileSize+2, dx

; Need to rewind file back to the beginning (seek to position zero):

                mov     bx, FileHandle  ;Need handle in BX.
                mov     ax, 4200h       ;Seek to beginning of file.
                xor     cx, cx          ;Seek to position zero
                xor     dx, dx
                int     21h
                jc      BadFile


; Okay, transmit the good stuff over to the receiving site:

                mov     al, byte ptr FileSize           ;Send the file
                call    SendByte                        ; size over.
                mov     al, byte ptr FileSize+1
                call    SendByte
                mov     al, byte ptr FileSize+2
                call    SendByte
                mov     al, byte ptr FileSize+3
                call    SendByte

                les     bx, FileNamePtr                 ;Send the characters
SendName:       mov     al, es:[bx]                     ; in the filename to
                call    SendByte                        ; the receiver until
                inc     bx                              ; we hit a zero byte.
                cmp     al, 0
                jne     SendName
                stc                                     ;Return success.
                ret

BadFile:        print
                byte    "Error transmitting file information:",0
                puti
                putcr
                clc
                ret
GetFileInfo     endp


; GetFileData-This procedure reads the data from the file and transmits
;             it to the receiver a byte at a time.

GetFileData     proc    near
                mov     ah, 3Fh         ;DOS read opcode.
                mov     cx, 512         ;Read 512 bytes at a time.
                mov     bx, FileHandle  ;File to read from.
                lea     dx, FileBuffer  ;Buffer to hold data.
                int     21h             ;Read the data
```

```
                jc      GFDError        ;Quit if error reading data.

                mov     cx, ax          ;Save # of bytes actually read.
                jcxz    GFDDone         ; quit if at EOF.
                lea     bx, FileBuffer  ;Send the bytes in the file
XmitLoop:       mov     al, [bx]        ; buffer over to the rcvr
                call    SendByte        ; one at a time.
                inc     bx
                loop    XmitLoop
                jmp     GetFileData     ;Read rest of file.

GFDError:       print
                byte    "DOS error #",0
                puti
                print
                byte    " while reading file",cr,lf,0
GFDDone:        ret
GetFileData     endp


; Okay, here's the main program that controls everything.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                meminit


; First, get the address of LPT1: from the BIOS variables area.

                mov     ax, 40h
                mov     es, ax
                mov     ax, es:[PrtrBase]
                mov     MyPortAdrs, ax

; See if we have a filename parameter:

                argc
                cmp     cx, 1
                je      GotName
                print
                byte    "Usage: transmit <filename>",cr,lf,0
                jmp     Quit


GotName:        call    Synchronize     ;Wait for the transmitter program.
                jnc     Quit

                call    GetFileInfo     ;Get file name and size.
                jnc     Quit

                call    GetFileData     ;Get the file's data.

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

Here is the receiver program that accepts and stores away the data sent by the program above:

```
; RECEIVE.ASM
;
; This program is the receiver portion of the programs that transmit files
; across a Laplink compatible parallel cable.
;
; This program assumes that the user want to use LPT1: for transmission.
; Adjust the equates, or read the port from the command line if this
; is inappropriate.

                .286
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


dseg            segment    para public 'data'

TimeOutConst    equ        100             ;About 1 min on 66Mhz 486.
PrtrBase        equ        8               ;Offset to LPT1: adrs.

MyPortAdrs      word       ?               ;Holds printer port address.
FileHandle      word       ?               ;Handle for output file.
FileBuffer      byte       512 dup (?)     ;Buffer for incoming data.

FileSize        dword      ?               ;Size of incoming file.
FileName        byte       128 dup (0)     ;Holds filename

dseg            ends

cseg            segment    para public 'code'
                assume     cs:cseg, ds:dseg


; TestAbort-    Reads the keyboard and gives the user the opportunity to
;               hit the ctrl-C key.

TestAbort       proc       near
                push       ax
                mov        ah, 1
                int        16h             ;See if keypress.
                je         NoKeypress
                mov        ah, 8           ;Read char, chk for ctrl-C
                int        21h
NoKeyPress:     pop        ax
                ret
TestAbort       endp



; GetByte-      Reads a single byte from the parallel port (four bits at
;               at time). Returns the byte in AL.

GetByte         proc       near
                push       cx
                push       dx

; Receive the L.O. Nibble.

                mov        dx, MyPortAdrs
                mov        al, 10h         ;Signal not busy.
                out        dx, al

                inc        dx              ;Point at status port

W4DLp:          mov        cx, 10000
Wait4Data:      in         al, dx          ;See if data available.
                test       al, 80h         ; (bit 7=0 if data available).
                loopne     Wait4Data
                je         DataIsAvail     ;Is data available?
                call       TestAbort       ;If not, check for ctrl-C.
```

```
                jmp         W4DLp

DataIsAvail:    shr         al, 3               ;Save this four bit package
                and         al, 0Fh             ; (This is the L.O. nibble
                mov         ah, al              ; for our byte).

                dec         dx                  ;Point at data register.
                mov         al, 0               ;Signal data taken.
                out         dx, al

                inc         dx                  ;Point at status register.
W4ALp:          mov         cx, 10000
Wait4Ack:       in          al, dx              ;Wait for transmitter to
                test        al, 80h             ; retract data available.
                loope       Wait4Ack            ;Loop until data not avail.
                jne         NextNibble          ;Branch if data not avail.
                call        TestAbort           ;Let user hit ctrl-C.
                jmp         W4ALp

; Receive the H.O. nibble:

NextNibble:     dec         dx                  ;Point at data register.
                mov         al, 10h             ;Signal not busy
                out         dx, al
                inc         dx                  ;Point at status port
W4D2Lp:         mov         cx, 10000
Wait4Data2:     in          al, dx              ;See if data available.
                test        al, 80h             ; (bit 7=0 if data available).
                loopne      Wait4Data2          ;Loop until data available.
                je          DataAvail2          ;Branch if data available.
                call        TestAbort           ;Check for ctrl-C.
                jmp         W4D2Lp

DataAvail2:     shl         al, 1               ;Merge this H.O. nibble
                and         al, 0F0h            ; with the existing L.O.
                or          ah, al              ; nibble.
                dec         dx                  ;Point at data register.
                mov         al, 0               ;Signal data taken.
                out         dx, al

                inc         dx                  ;Point at status register.
W4A2Lp:         mov         cx, 10000
Wait4Ack2:      in          al, dx              ;Wait for transmitter to
                test        al, 80h             ; retract data available.
                loope       Wait4Ack2           ;Wait for data not available.
                jne         ReturnData          ;Branch if ack.
                call        TestAbort           ;Check for ctrl-C
                jmp         W4A2Lp

ReturnData:     mov         al, ah              ;Put data in al.
                pop         dx
                pop         cx
                ret
GetByte         endp



; Synchronize- This procedure waits until it sees all zeros on the input
;              bits we receive from the transmitting site. Once it receives
;              all zeros, it writes all ones to the output port. When
;              all ones come back, it writes all zeros. It repeats this
;              process until the transmitting site writes the value 05h.

Synchronize     proc        near

                print
                byte        "Synchronizing with transmitter program"
                byte        cr,lf,0

                mov         dx, MyPortAdrs
                mov         al, 0               ;Initialize our output port
                out         dx, al              ; to prevent confusion.
                mov         bx, TimeOutConst    ;Time out condition.
```

```
SyncLoop:     mov       cx, 0             ;For time out purposes.
SyncLoop0:    inc       dx                ;Point at input port.
              in        al, dx            ;Read our input bits.
              dec       dx
              and       al, 78h           ;Keep only the data bits.
              cmp       al, 78h           ;Check for all ones.
              je        Got1s             ;Branch if all ones.
              cmp       al, 0             ;See if all zeros.
              loopne    SyncLoop0

; Since we just saw a zero, write all ones to the output port.

              mov       al, 0FFh      ;Write all ones
              out       dx, al

; Now wait for all ones to arrive from the transmitting site.

SyncLoop1:    inc       dx                ;Point at status register.
              in        al, dx            ;Read status port.
              dec       dx                ;Point back at data register.
              and       al, 78h           ;Keep only the data bits.
              cmp       al, 78h           ;Are they all ones?
              loopne    SyncLoop1         ;Repeat while not ones.
              je        Got1s             ;Branch if got ones.

; If we've timed out, check to see if the user has pressed ctrl-C to
; abort.

              call      TestAbort         ;Check for ctrl-C.
              dec       bx                ;See if we've timed out.
              jne       SyncLoop          ;Repeat if time-out.

              print
              byte      "Receive: connection timed out during synchronization"
              byte      cr,lf,0
              clc                         ;Signal time-out.
              ret

; Jump down here once we've seen both a zero and a one. Send the two
; in combinations until we get a 05h from the transmitting site or the
; user presses Ctrl-C.

Got1s:        inc       dx                ;Point at status register.
              in        al, dx            ;Just copy whatever appears
              dec       dx                ; in our input port to the
              shr       al, 3             ; output port until the
              and       al, 0Fh           ; transmitting site sends
              cmp       al, 05h           ; us the value 05h
              je        Synchronized
              not       al                ;Keep inverting what we get
              out       dx, al            ; and send it to xmitter.
              call      TestAbort         ;Check for CTRL-C here.
              jmp       Got1s


; Okay, we're synchronized. Return to the caller.

Synchronized:
              and       al, 0Fh           ;Make sure busy bit is one
              out       dx, al            ; (bit 4=0 for busy=1).
              print
              byte      "Synchronized with transmitting site"
              byte      cr,lf,0
              stc
              ret
Synchronize   endp


; GetFileInfo- The transmitting program sends us the file length and a
;              zero terminated filename. Get that data here.

GetFileInfo   proc      near
              mov       dx, MyPortAdrs
              mov       al, 10h           ;Set busy bit to zero.
```

```
                out     dx, al          ;Tell xmit pgm, we're ready.

; First four bytes contain the filesize:

                call    GetByte
                mov     byte ptr FileSize, al
                call    GetByte
                mov     byte ptr FileSize+1, al
                call    GetByte
                mov     byte ptr FileSize+2, al
                call    GetByte
                mov     byte ptr FileSize+3, al

; The next n bytes (up to a zero terminating byte) contain the filename:

                mov     bx, 0
GetFileName:    call    GetByte
                mov     FileName[bx], al
                call    TestAbort
                inc     bx
                cmp     al, 0
                jne     GetFileName

                ret
GetFileInfo     endp


; GetFileData- Receives the file data from the transmitting site
;              and writes it to the output file.

GetFileData     proc    near

; First, see if we have more than 512 bytes left to go

                cmp     word ptr FileSize+2, 0          ;If H.O. word is not
                jne     MoreThan512                     ; zero, more than 512.
                cmp     word ptr FileSize, 512          ;If H.O. is zero, just
                jbe     LastBlock                       ; check L.O. word.

; We've got more than 512 bytes left to go in this file, read 512 bytes
; at this point.

MoreThan512:    mov     cx, 512                         ;Receive 512 bytes
                lea     bx, FileBuffer                  ; from the xmitter.
ReadLoop:       call    GetByte                         ;Read a byte.
                mov     [bx], al                        ;Save the byte away.
                inc     bx                              ;Move on to next
                loop    ReadLoop                        ; buffer element.

; Okay, write the data to the file:

                mov     ah, 40h                         ;DOS write opcode.
                mov     bx, FileHandle                  ;Write to this file.
                mov     cx, 512                         ;Write 512 bytes.
                lea     dx, Filebuffer                  ;From this address.
                int     21h
                jc      BadWrite                        ;Quit if error.

; Decrement the file size by 512 bytes:

                sub     word ptr FileSize, 512          ;32-bit subtraction
                sbb     word ptr FileSize, 0            ; of 512.
                jmp     GetFileData

; Process the last block, that contains 1..511 bytes, here.

LastBlock:
                mov     cx, word ptr FileSize           ;Receive the last
                lea     bx, FileBuffer                  ; 1..511 bytes from
ReadLB:         call    GetByte                         ; the transmitter.
                mov     [bx], al
                inc     bx
                loop    ReadLB
```

```
                mov     ah, 40h                         ;Write the last block
                mov     bx, FileHandle                  ; of bytes to the
                mov     cx, word ptr FileSize           ; file.
                lea     dx, Filebuffer
                int     21h
                jnc     Closefile

BadWrite:       print
                byte    "DOS error #",0
                puti
                print
                byte    " while writing data.",cr,lf,0

; Close the file here.

CloseFile:      mov     bx, FileHandle                  ;Close this file.
                mov     ah, 3Eh                         ;DOS close opcode.
                int     21h
                ret
GetFileData     endp


; Here's the main program that gets the whole ball rolling.

Main            proc
                mov     ax, dseg
                mov     ds, ax
                meminit


; First, get the address of LPT1: from the BIOS variables area.

                mov     ax, 40h         ;Point at BIOS variable segment.
                mov     es, ax
                mov     ax, es:[PrtrBase]
                mov     MyPortAdrs, ax

                call    Synchronize     ;Wait for the transmitter program.
                jnc     Quit

                call    GetFileInfo     ;Get file name and size.

                printf
                byte    "Filename: %s\nFile size: %ld\n",0
                dword   Filename, FileSize

                mov     ah, 3Ch         ;Create file.
                mov     cx, 0           ;Standard attributes
                lea     dx, Filename
                int     21h
                jnc     GoodOpen
                print
                byte    "Error opening file",cr,lf,0
                jmp     Quit

GoodOpen:       mov     FileHandle, ax
                call    GetFileData     ;Get the file's data.

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

21.5 Summary

The PC's parallel port, though originally designed for controlling parallel printers, is a general purpose eight bit output port with several handshaking lines you can use to control many other devices in addition to printers.

In theory, parallel communications should be many times faster than serial communications. In practice, however, real world constraints and economics prevent this from being the case. Nevertheless, you can still connect high performance devices to the PC's parallel port.

The PC's parallel ports come in two varieties: unidirectional and bidirectional. The bidirectional versions are available only on PS/2s, certain laptops, and a few other machines. Whereas the eight data lines are output only on the unidirectional ports, you can program them as inputs or outputs on the bidirectional port. While this bidirectional operation is of little value to a printer, it can improve the performance of other devices that connect to the parallel port, such as disk and tape drives, network adapters, SCSI adapters, and so on.

When the system communicates with some other device over the parallel port, it needs some way to tell that device that data is available on the data lines. Likewise, the devices needs some way to tell the system that it is not busy and it has accepted the data. This requires some additional signals on the parallel port known as handshaking lines. A typical PC parallel port provides three handshaking signals: the data available strobe, the data taken acknowledge signal, and the device busy line. These lines easily control the flow of data between the PC and some external device.

In addition to the handshaking lines, the PC's parallel port provides several other auxiliary I/O lines as well. In total, there are 12 output lines and five input lines on the PC's parallel port. There are three I/O ports in the PC's address space associated with each I/O port. The first of these (at the port's base address) is the data register. This is an eight bit output register on unidirectional ports, it is an input/output register on bidirectional ports. The second register, at the base address plus one, is the status register. The status register is an input port. Five of those bits correspond to the five input lines on the PC's parallel port. The third register (at base address plus two) is the control register. Four of these bits correspond to the additional four output bits on the PC, one of the bits controls the IRQ line on the parallel port, and a sixth bit controls the data direction on the birdirectional ports.

For more information on the parallel port's hardware configuration, see:

- "Basic Parallel Port Information" on page 1199
- "The Parallel Port Hardware" on page 1201

Although many vendors use the parallel port to control lots of different devices, a parallel printer is still the device most often connected to the parallel port. There are three ways application programs commonly send data to the printer: by calling DOS to print a character, by calling BIOS' int 17h ISR to print a character, or by talking directly to the parallel port. You should avoid this last technique because of possible software incompatibilities with other devices that connect to the parallel port. For more information on printing data, including how to write your own int 17h ISR/printer driver, see:

- "Controlling a Printer Through the Parallel Port" on page 1202
- "Printing via DOS" on page 1203
- "Printing via BIOS" on page 1203
- "An INT 17h Interrupt Service Routine" on page 1203

One popular use of the parallel port is to transfer data between two computers; for example, transferring data between a desktop and a laptop machine. To demonstrate how to use the parallel port to control other devices besides printers, this chapter presents a program to transfer data between computers on the unidirectional parallel ports (it also works on bidirectional ports). For all the details, see

- "Inter-Computer Communications on the Parallel Port" on page 1209

The RS-232 serial communication standard is probably the most popular serial communication scheme in the world. Although it suffers from many drawbacks, speed being the primary one, it use is widespread and there are literally thousands of devices you can connect to a PC using an RS-232 interface. The PC supports up to four RS-232 compatible devices using the COM1:, COM2:, COM3:, and COM4: devices[1]. For those who need even more serial devices (e.g., to control an electronic bulletin board system [BBS], you can even buy devices that let you add 16, or more, serial ports to the PC. Since most PCs only have one or two serial ports, we will concentrate on how to use COM1: and COM2: in this chapter.

Although, in theory, the PC's original design allows system designers to implement the serial communication ports using any hardware they desire, much of today's software that does serial communication talks directly to the 8250 Serial Communications Chip (SCC) directly. This introduces the same compatibility problems you get when you talk directly to the parallel port hardware. However, whereas the BIOS provides an excellent interface to the parallel port, supporting anything you would wish to do by going directly to the hardware, the serial support is not so good. Therefore, it is common practice to bypass the BIOS int 14h functions and control the 8250 SCC chip directly so software can access every bit of every register on the 8250.

Perhaps an even greater problem with the BIOS code is that it does not support interrupts. Although software controlling parallel ports rarely uses interrupt driven I/O[2], it is very common to find software that provides interrupt service routines for the serial ports. Since the BIOS does not provide such routines, any software that wants to use interrupt driven serial I/O will need to talk directly to the 8250 and bypass BIOS anyway. Therefore, the first part of this chapter will discuss the 8250 chip.

Manipulating the serial port is not difficult. However, the 8250 SCC contains lots of registers and provides many features. Therefore it takes a lot of code to control every feature of the chip. Fortunately, you do not have to write that code yourself. The UCR Standard Library provides an excellent set of routines that let you control the 8250. They even an interrupt service routine allowing interrupt driven I/O. The second part of this chapter will present the code from the Standard Library as an example of how to program each of the registers on the 8250 SCC.

22.1 The 8250 Serial Communications Chip

The 8250 and compatible chips (like the 16450 and 16550 devices) provide nine I/O registers. Certain upwards compatible devices (e.g., 16450 and 16550) provide a tenth register as well. These registers consume eight I/O port addresses in the PC's address space. The hardware and locations of the addresses for these devices are the following:

Table 81: COM Port Addresses

Port	Physical Base Address (in hex)	BIOS variable Containing Physical Address[a]
COM1:	3F8	40:0
COM2:	2F8	40:2

a. Locations 40:4 and 40:6 contain the logical addresses for COM3: and COM4:, but we will not consider those ports here.

1. Most programs support only COM1: and COM2:. Support for additional serial devices is somewhat limited among various applications.
2. Because many parallel port adapters do not provide hardware support for interrupts.

Like the PC's parallel ports, we can swap COM1: and COM2: at the software level by swapping their base addresses in BIOS variable 40:0 and 40:2. However, software that goes directly to the hardware, especially interrupt service routines for the serial ports, needs to deal with hardware addresses, not logical addresses. Therefore, we will always mean I/O base address 3F8h when we discuss COM1: in this chapter. Likewise, we will always mean I/O base address 2F8h when we discuss COM2: in this chapter.

The base address is the first of eight I/O locations consumed by the 8250 SCC. The exact purpose of these eight I/O locations appears in the following table:

Table 82: 8250 SCC Registers

I/O Address (hex)	Description
3F8/2F8	Receive/Transmit data register. Also the L.O. byte of the Baud Rate Divisor Latch register.
3F9/2F9	Interrupt Enable Register. Also the H.O. byte of the Baud Rate Divisor Register.
3FA/2FA	Interrupt Identification Register (read only).
3FB/2FB	Line Control Register.
3FC/2FC	Modem Control Register.
3FD/2FD	Line Status Register (read only).
3FE/2FE	Modem Status Register (read only).
3FF/2FF	Shadow Receive Register (read only, not available on original PCs).

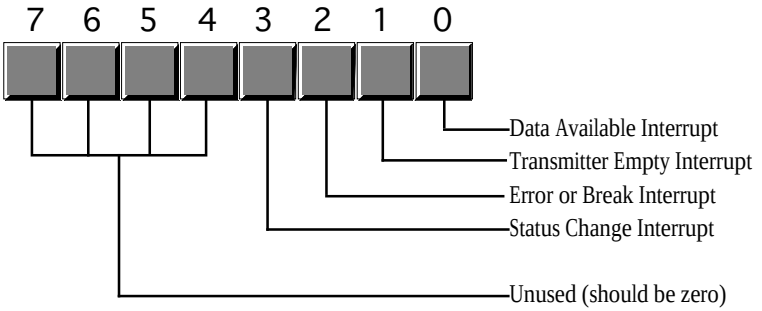
The following sections describe the purpose of each of these registers.

22.1.1 The Data Register (Transmit/Receive Register)

The data register is actually two separate registers: the transmit register and the receive register. You select the transmit register by writing to I/O addresses 3F8h or 2F8h, you select the receive register by reading from these addresses. Assuming the transmit register is empty, writing to the transmit register begins a data transmission across the serial line. Assuming the receive register is full, reading the receive register returns the data. To determine if the transmitter is empty or the receiver is full, see the Line Status Register. Note that the Baud Rate Divisor register shares this I/O address with the receive and transmit registers. Please see “The Baud Rate Divisor” on page 1225 and “The Line Control Register” on page 1227 for more information on the dual use of this I/O location.

22.1.2 The Interrupt Enable Register (IER)

When operating in interrupt mode, the 8250 SCC provides four sources of interrupt: the character received interrupt, the transmitter empty interrupt, the communication error interrupt, and the status change interrupt. You can individually enable or disable these interrupt sources by writing ones or zeros to the 8250 IER (Interrupt Enable Register). Writing a zero to a corresponding bit disables that particular interrupt. Writing a one enables that interrupt. This register is read/write, so you can interrogate the current settings at any time (for example, if you want to mask in a particular interrupt without affecting the others). The layout of this register is



Serial Port Interrupt Enable Register (IER)

The interrupt enable register I/O location is also common with the Baud Rate Divisor Register. Please see the next section and “The Line Control Register” on page 1227 for more information on the dual use of this I/O location.

22.1.3 The Baud Rate Divisor

The Baud Rate Divisor Register is a 16 bit register that shares I/O locations 3F8h/2F8h and 3F9h/2F9h with the data and interrupt enable registers. Bit seven of the Line Control Register (see “The Line Control Register” on page 1227) selects the divisor register or the data/interrupt enable registers.

The Baud Rate Divisor register lets you select the data transmission rate (properly called *bits per second*, or *bps*, not baud[3]). The following table lists the values you should write to these registers to control the transmission/reception rate:

Table 83: Baud Rate Divisor Register Values

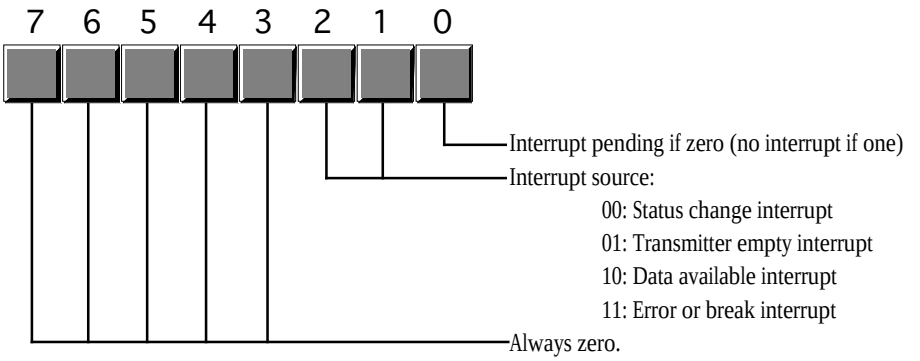
Bits Per Second	3F9/3F9 Value	3F8/2F8 Value
110	4	17h
300	1	80h
600	0	C0h
1200	0	60h
1800	0	40h
2400	0	30h
3600	0	20h
4800	0	18h
9600	0	0Ch
19.2K	0	6
38.4K	0	3
56K	0	1

3. The term “baud” describes the rate at which tones can change on a modem/telephone line. It turns out that, with normal telephone lines, the maximum baud rate is 600 baud. Modems that operate at 1200 bps use a different technique (beyond switching tones) to increase the data transfer rate. In general, there is no such thing as a “1200 baud,” “9600 baud,” or “14.4 kbaud” modem. Properly, these are 1200 bps, 9600bps, and 14.4K bps modems.

You should only operate at speeds greater than 19.2K on fast PCs with high performance SCCs (e.g., 16450 or 16550). Furthermore, you should use high quality cables and keep your cables very short when running at high speeds.

22.1.4 The Interrupt Identification Register (IIR)

The Interrupt Identification Register is a read-only register that specifies whether an interrupt is pending and which of the four interrupt sources requires attention. This register has the following layout:



Interrupt Identification Register (IIR)

Since the IIR can only report one interrupt at a time, and it is certainly possible to have two or more pending interrupts, the 8250 SCC prioritizes the interrupts. Interrupt source 00 (status change) has the lowest priority and interrupt source 11 (error or break) has the highest priority; i.e., the interrupt source number provides the priority (with three being the highest priority).

The following table describes the interrupt sources and how you "clear" the interrupt value in the IIR. If two interrupts are pending and you service the higher priority request, the 8250 SCC replaces the value in the IIR with the identification of the next highest priority interrupt source.

Table 84: Interrupt Cause and Release Functions

Priority	ID Value	Interrupt	Caused By	Reset By
Highest	11b	Error or Break	Overrun error, parity error, framing error, or break interrupt.	Reading the Line Status Register.
Next to highest	10b	Data available	Data arriving from an external source in the Receive Register.	Reading the Receive Register.
Next to lowest	01b	Transmitter empty	The transmitter finishes sending data and is ready to accept additional data.	Reading the IIR (with an interrupt ID of 01b) or writing to the Data Register.
Lowest	00b	Modem Status	Change in clear to send, data set ready, ring indicator, or received line signal detect signals.	Reading the modem status register.

One interesting point to note about the organization of the IIR: the bit layout provides a convenient way to transfer control to the appropriate section of the SCC interrupt service routine. Consider the following code:

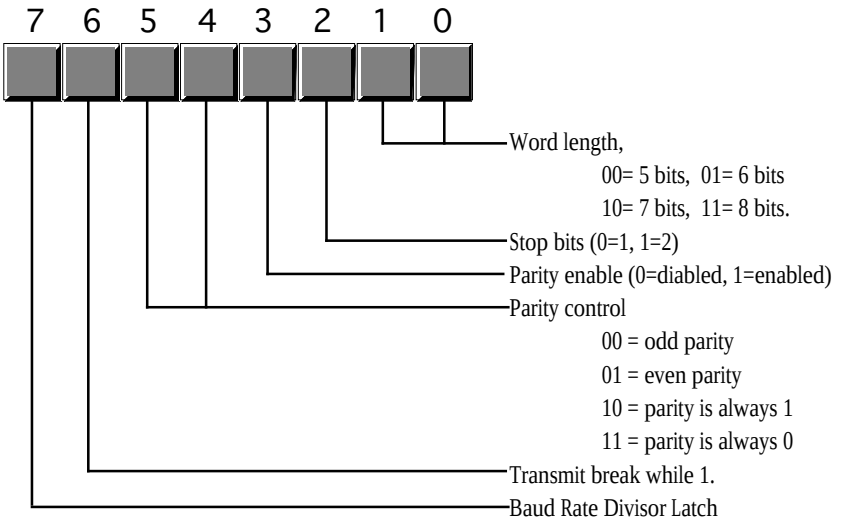
```
        .
        .
        .
in      al, dx          ;Read IIR.
```

```
                mov         bl, al
                mov         bh, 0
                jmp         HandlerTbl[bx]
HandlerTbl      word        RLSHandler, RDHandler, TEHandler, MSHandler
```

When an interrupt occurs, bit zero of the IIR will be zero. The next two bits contain the interrupt source number and the H.O. five bits are all zero. This lets us use the IIR value as the index into a table of pointers to the appropriate handler routines, as the above code demonstrates.

22.1.5 The Line Control Register

The Line Control Register lets you specify the transmission parameters for the SCC. This includes setting the data size, number of stop bits, parity, forcing a break, and selecting the Baud Rate Divisor Register (see “The Baud Rate Divisor” on page 1225). The Line Control Register is laid out as follows:



Line Control Register (LCR)

The 8250 SCC can transmit serial data as groups of five, six, seven, or eight bits. Most modern serial communication systems use seven or eight bits for transmission (you only need seven bits to transmit ASCII, eight bits to transmit binary data). By default, most applications transmit data using eight data bits. Of course, you always read eight bits from the receive register; the 8250 SCC pads all H.O. bits with zero if you are receiving less than eight bits. Note that if you are only transmitting ASCII characters, the serial communications will run about 10% faster with seven bit transmission rather than with eight bit transmission. This is an important thing to keep in mind if you control both ends of the serial cable. On the other hand, you will usually be connecting to some device that has a fixed word length, so you will have to program the SCC specifically to match that device.

A serial data transmission consists of a *start bit*, five to eight *data bits*, and one or two *stop bits*. The start bit is a special signal that informs the SCC (or other device) that data is arriving on the serial line. The stop bits are, essentially, the absence of a start bit to provide a small amount of time between the arrival of consecutive characters on the serial line. By selecting two stop bits, you insert some additional time between the transmission of each character. Some older devices may require this additional time or they will get confused. However, almost all modern serial devices are perfectly happy with a single stop bit. Therefore, you should usually program the chip with only one stop bit. Adding a second stop bit increases transmission time by about 10%.

The parity bits let you enable or disable parity and choose the type of parity. Parity is an error detection scheme. When you enable parity, the SCC adds an extra bit (the parity bit) to the transmission. If you select odd parity, the parity bit contains a zero or one so that the L.O. bit of the sum of the data and parity

bits is one. If you select even parity, the SCC produces a parity bit such that the L.O. bit of the sum of the parity and data bits is zero. The “stuck parity” values (10b and 11b) always produce a parity bit of zero or one. The main purpose of the parity bit is to detect a possible transmission error. If you have a long, noisy, or otherwise bad serial communications channel, it is possible to lose information during transmission. When this happens, it is unlikely that the sum of the bits will match the parity value. The receiving site can detect this “parity error” and report the error in transmission.

You can also use the stuck parity values (10b and 11b) to strip the eighth bit and always replace it with a zero or one during transmission. For example, when transmitting eight bit PC/ASCII characters to a different computer system it is possible that the PC's extended character set (those characters whose code is 128 or greater) does not map to the same character on the destination machine. Indeed, sending such characters may create problems on that machine. By setting the word size to seven bits and the parity to enabled and stuck at zero, you can automatically strip out all H.O. bits during transmission, replacing them with zero. Of course, if any extended characters come along, the SCC will map them to possibly unrelated ASCII characters, but this is a useful trick, on occasion.

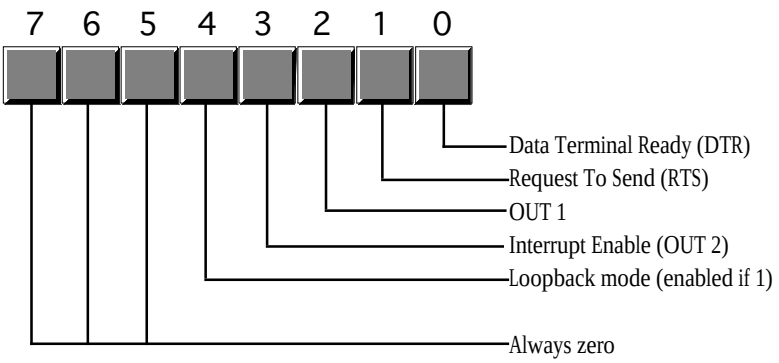
The break bit transmits a break signal to the remote system as long as there is a one programmed in this bit position. You should not leave break enabled while trying to transmit data. The break signal comes from the teletype days. A break is similar to ctrl-C or ctrl-break on the PC's keyboard. It is supposed to interrupt a program running on a remote system. Note that the SCC can detect an incoming break signal and generate an appropriate interrupt, but this break signal is coming from the remote system, it is not (directly) connected to the outgoing break signal the LCR controls.

Bit seven of the LCR is the Baud Rate Divisor Register latch bit. When this bit contains a one, locations 3F8h/2F8h and 3F9h/2F9h become the Baud Rate Divisor Register. When this bit contains a zero, those I/O locations correspond to the Data Registers and the Interrupt Enable Registers. You should always program this bit with a zero except while initializing the speed of the SCC.

The LCR is a read/write register. Reading the LCR returns the last value written to it.

22.1.6 The Modem Control Register

The 8250’s Modem Control Register contains five bits that let you directly control various output pins on the 8250 as well as enable the 8250's *loopback* mode. The following diagram displays the contents of this register:



Modem Control Register (MCR)

The 8250 routes the DTR and RTS bits directly to the DTR and RTS lines on the 8250 chip. When these bits are one, the corresponding outputs are active[4]. These lines are two separate handshake lines for RS-232 communications.

4. It turns out that the DTR and RTS lines are active low, so the 8250 actually inverts these lines on their way out. However, the receiving site reinverts these lines so the receiving site (if it is an 8250 SCC) will read these bits as one when they are active. See the description of the line status register for details.

The DTR signal is comparable to a *busy* signal. When a site's DTR line is inactive, the other site is not supposed to transmit data to it. The DTR line is a *manual* handshake line. It appears as the Data Set Ready (DSR) line on the other side of the serial cable. The other device must explicitly check its DSR line to see if it can transmit data. The DTR/DSR scheme is mainly intended for handshaking between computers and modems.

The RTS line provides a second form of handshake. It's corresponding input signal is CTS (Clear To Send). The RTS/CTS handshake protocol is mainly intended for directly connected devices like computers and printers. You may ask "why are there two separate, but orthogonal handshake protocols?" The reason is because RS-232C has developed over the last 100 years (from the days of the first telegraphs) and is the result of combining several different schemes over the years.

Out1 is a general purpose output on the SCC that has very little use on the IBM PC. Some adapter boards connect this signal, other leave it disconnected. In general, this bit has no function on PCs.

The Interrupt Enable bit is a PC-specific item. This is normally a general purpose output (OUT 2) on the 8250 SCC. However, IBM's designers connected this output to an external gate to enable or disable all interrupts from the SCC. This bit must be programmed with a one to enable interrupts. Likewise, you must ensure that this bit contains a zero if you are not using interrupts.

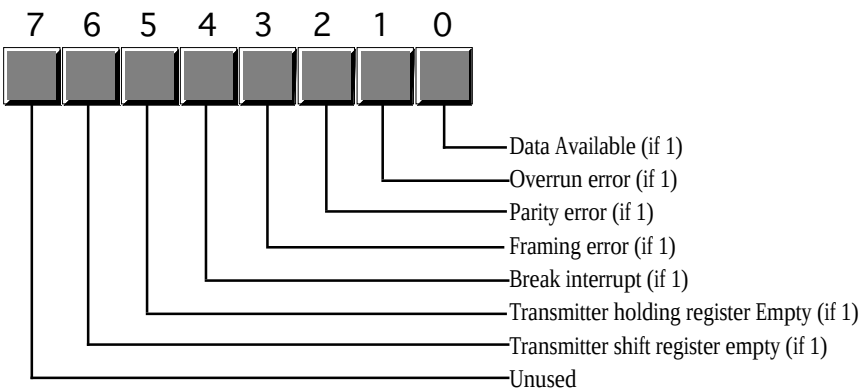
The loopback bit connects the transmitter register to the receive register. All data sent out the transmitter immediately comes back in the receive register. This is useful for diagnostics, testing software, and detecting the serial chip. Note, unfortunately, that the loopback circuit will not generate any interrupts. You can only use this technique with polled I/O.

The remaining bits in the MCR are reserved should always contain zero. Future versions of the SCC (or compatible chips) may use these bits for other purposes, with zero being the default (8250 simulation) state.

The MCR is a read/write register. Reading the MCR returns the last value written to it.

22.1.7 The Line Status Register (LSR)

The Line Status Register (LSR) is a read-only register that returns the current communication status. The bit layout for this register is the following:



Line Status Register (LSR)

The data available bit is set if there is data available in the Receive Register. This also generates an interrupt. Reading the data in the Receive Register clears this bit.

The 8250 Receive Register can only hold one byte at a time. If a byte arrives and the program does not read it and then a second byte arrives, the 8250 wipes out the first byte with the second. The 8250 SCC sets

the overrun error bit when this occurs. Reading the LSR clears this bit (after reading the LSR). This error will generate the high priority error interrupt.

The 8250 sets the parity bit if it detects a parity error when receiving a byte. This error only occurs if you have enabled the parity operation in the LCR. The 8250 resets this bit after you read the LSR. When this error occurs, the 8250 will generate the error interrupt.

Bit three is the framing error bit. A framing error occurs if the 8250 receives a character without a valid stop bit. The 8250 will clear this bit after you read the LSR. This error will generate the high priority error interrupt.

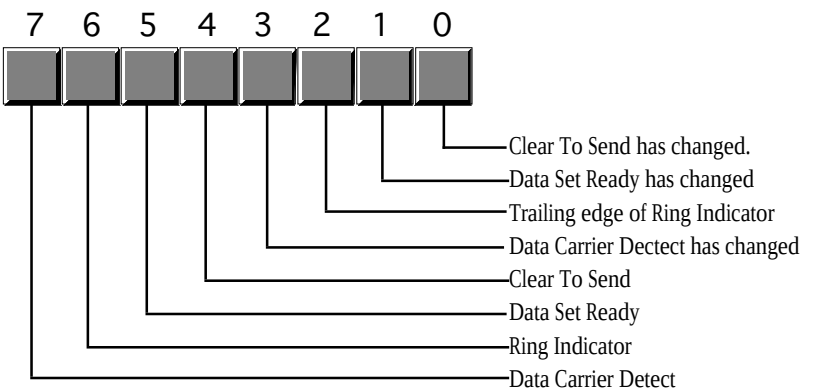
The 8250 sets the break interrupt bit when it receives the break signal from the transmitting device. This will also generate an error interrupt. Reading the LSR clears this bit.

The 8250 sets bit five, the transmitter holding register empty bit, when it is okay to write another character to the Data Register. Note that the 8250 actually has two registers associated with the transmitter. The transmitter shift register contains the data actually being shifted out over the serial line. The transmitter holding register holds a value that the 8250 writes to the shift register when it finishes shifting out a character. Bit five indicates that the holding register is empty and the 8250 can accept another byte. Note that the 8250 might still be shifting out a character in parallel with this operation. The 8250 can generate an interrupt when the transmitter holding register is empty. Reading the LSR or writing to the Data Register clears this bit.

The 8250 sets bit six when both the transmitter holding and transmitter shift registers are empty. This bit is clear when either register contains data.

22.1.8 The Modem Status Register (MSR)

The Modem Status Register (MSR) reports the status of the handshake and other modem signals. Four bits provide the instantaneous values of these signals, the 8250 sets the other four bits if any of these signals change since the last time the CPU interrogates the MSR. The MSR has the following layout:



Modem Status Register (MSR)

The Clear To Send bit (bit #4) is a handshaking signal. This is normally connected to the RTS (Request To Send) signal on the remove device. When that remote device asserts its RTS line, data transmission can take place.

The Data Set Ready bit (bit #5) is one if the remote device is not busy. This input is generally connected to the Data Terminal Ready (DTR) line on the remote device.

The 8250 chip sets the Ring Indicator bit (bit #6) when the modem asserts the ring indicator line. You will rarely use this signal unless you are writing modem controlling software that automatically answers a telephone call.

The Data Carrier Detect bit (DCD, bit #7) is another modem specific signal. This bit contains a one while the modem detects a carrier signal on the phone line.

Bits zero through three of the MSR are the “delta” bits. These bits contain a one if their corresponding modem status signal changes. Such an occurrence will also generate a modem status interrupt. Reading the MSR will clear these bits.

22.1.9 The Auxiliary Input Register

The auxiliary input register is available only on later model 8250 compatible devices. This is a read-only register that returns the same value as reading the data register. The difference between reading this register and reading the data register is that reading the auxiliary input register does not affect the data available bit in the LSR. This allows you to test the incoming data value without removing it from the input register. This is useful, for example, when chaining serial chip interrupt service routines and you want to handle certain “hot” values in one ISR and pass all other characters on to a different serial ISR.

22.2 The UCR Standard Library Serial Communications Support Routines

Although programming the 8250 SCC doesn’t seem like a real big problem, invariably it is a difficult chore (and tedious) to write all the software necessary to get the serial communication system working. This is especially true when using interrupt driven serial I/O. Fortunately, you do not have to write this software from scratch, the UCR Standard library provides 21 support routines that trivialize the use of the serial ports on the PC. About the only drawback to these routines is that they were written specifically for COM1:, although it isn’t too much work to modify them to work with COM2:. The following table lists the available routines:

Table 85: Standard Library Serial Port Support

Name	Inputs	Outputs	Description
ComBaud	AX: bps (baud rate) = 110, 150, 300, 600, 1200, 2400, 4800, 9600, or 19200		Sets the communication rate for the serial port. ComBaud only supports the specified speeds. If ax contains some other value on entry, ComBaud ignores the value.
ComStop	AX: 1 or 2		Sets the number of stop bits. The ax register contains the number of stop bits to use (1 or 2).
ComSize	AX: word size (5, 6, 7, or 8)		Sets the number of data bits. The ax register contains the number of bits to transmit for each byte on the serial line.
ComParity	AX: Parity selector. If bit zero is zero, parity off, if bit zero is one, bits one and two are: 00 - odd parity 01 - even parity 10 - parity stuck at 0 11 - parity stuck at 1		Sets the parity (if any) for the serial communications.
ComRead		AL- Character read from port.	Waits until a character is available from in the data register and returns that character. Used for polled I/O on the serial port. Do not use if you’ve activated the serial interrupts (see ComInitIntr).

Table 85: Standard Library Serial Port Support

Name	Inputs	Outputs	Description
ComWrite	AL- Character to write.		Waits until the transmitter holding register is empty, then writes the character in al to the output register. Used for polled I/O on the serial port. Do not use with interrupts activated.
ComTstIn		AL=0 if no character, AL=1 if char avail.	Test to see if a character is available at the serial port. Use only for polling I/O, do not use with interrupts activated.
ComTstOut		AL=0 if transmitter busy, AL=1 if not busy.	Test to see if it is okay to write a character to the output register. Use with polled I/O only, do not use with interrupts active.
ComGetLSR		AL= Current LSR value.	Returns the current LSR value in the al register. See the section on the LSR for more details.
ComGetMSR		AL= Current MSR Value.	Returns the current MSR value in the al register. See the section on the MSR for more details.
ComGetMCR		AL= Current MCR Value.	Returns the current MCR value in the al register. See the section on the MCR for more details.
ComSetMCR	AL = new MCR Value		Stores the value in al into the MCR register. See the section on the MCR for more details.
ComGetLCR		AL= Current LCR Value.	Returns the current LCR value in the al register. See the section on the LCR for more details.
ComSetLCR	AL = new LCR Value		Stores the value in al into the LCR register. See the section on the LCR for more details.
ComGetIIR		AL= Current IIR Value.	Returns the current IIR value in the al register. See the section on the IIR for more details.
ComGetIER		AL= Current IER Value.	Returns the current IER value in the al register. See the section on the IER for more details.
ComSetIER	AL = new IER Value		Stores the value in al into the IER register. See the section on the IER for more details.
ComInitIntr			Initializes the system to support interrupt driven serial I/O. See details below.
ComDisIntr			Resets the system back to polled serial I/O
ComIn			Reads a character from the serial port when operating with interrupt driven I/O.
ComOut			Writes a character to the serial port using interrupt driven I/O.

The interrupt driven I/O features of the Standard Library routines deserve further explanation. When you call the ComInitIntr routine, it patches the COM1: interrupt vectors (int 0Ch), enables IRQ 4 in the 8259A PIC, and enables read and write interrupts on the 8250 SCC. One thing this call does not do that you should is patch the break and critical error exception vectors (int 23h and int 24h) to handle any program aborts that come along. When your program quits, either normally or via one of the above exceptions, it must call ComDisIntr to disable the interrupts. Otherwise, the next time a character arrives at the serial port the machine may crash since it will attempt to jump to an interrupt service routine that might not be there anymore.

The ComIn and ComOut routines handle interrupt driven serial I/O. The Standard Library provides a reasonable input and output buffer (similar to the keyboard's type ahead buffer), so you do not have to worry about losing characters unless your program is really, really slow or rarely reads any data from the serial port.

Between the ComInitIntr and ComDisIntr calls, you should not call any other serial support routines except ComIn and ComOut. The other routines are intended for polled I/O or initialization. Obviously, you should do any necessary initialization before enabling interrupts, and there is no need to do polled I/O while the interrupts are operational. Note that there is no equivalent to ComTstIn and ComTstOut while operating in interrupt mode. These routines are easy to write, instructions appear in the next section.

22.3 Programming the 8250 (Examples from the Standard Library)

The UCR Standard Library Serial Communication routines provide an excellent example of how to program the 8250 SCC directly, since they use nearly all the features of that chip on the PC. Therefore, this section will list each of the routines and describe exactly what that routine is doing. By studying this code, you can learn about all the details associated with the SCC and discover how to extend or otherwise modify the Standard Library routines.

```
; Useful equates:

BIOSvars      =         40h           ;BIOS segment address.
Com1Adrs      =         0             ;Offset in BIOS vars to COM1: address.
Com2Adrs      =         2             ;Offset in BIOS vars to COM2: address.

BufSize       =         256              ;# of bytes in buffers.


; Serial port equates. If you want to support COM2: rather than COM1:, simply
; change the following equates to 2F8h, 2F9h, ...

ComPort       =         3F8h
ComIER        =         3F9h
ComIIR        =         3FAh
ComLCR        =         3FBh
ComMCR        =         3FCh
ComLSR        =         3FDh
ComMSR        =         3FEh


; Variables, etc. This code assumes that DS=CS. That is, all the variables
; are in the code segment.
;
; Pointer to interrupt vector for int 0Ch in the interrupt vector table.
; Note: change these values to 0Bh*4 and 0Bh*4 + 2 if you want to support
; the COM2: pot.

int0Cofs equ   es:[0Ch*4]
int0Cseg equ   es:[0Ch*4 + 2]

OldInt0c      dword     ?

; Input buffer for incoming character (interrupt operation only). See the
; chapter on data structures and the description of circular queus for
; details on how this buffer works. It operates in a fashion not unlike
; the keyboard's type ahead buffer.

InHead        word      InpBuf
InTail        word      InpBuf
InpBuf        byte      Bufsize dup (?)
InpBufEnd     equ       this byte

; Output buffer for characters waiting to transmit.

OutHead       word      OutBuf
OutTail       word      OutBuf
OutBuf        byte      BufSize dup (?)
OutBufEnd     equ       this byte

; The i8259a variable holds a copy of the PIC's IER so we can restore it
; upon removing our interrupt service routines from memory.
```

```
i8259a          byte        0               ;8259a interrupt enable register.

; The TestBuffer variable tells us whether we have to buffer up characters
; or if we can store the next character directly into the 8250's output
; register (See the ComOut routine for details).

TestBuffer      db          0
```

The first set of routines provided by the Standard Library let you initialize the 8250 SCC. These routines provide "programmer friendly" interfaces to the baud rate divisor and line control registers. They let you set the baud rate, data size, number of stop bits, and parity options on the SCC.

The ComBaud routine sets the 8250's transfer rate (in bits per second). This routine provides a nice "programmer's interface" to the 8250 SCC. Rather than having to compute the baud rate divisor value yourself, you can simply load ax with the bps value you want and simply call this routine. Of course, one problem is that you must choose a bps value that this routine supports or it will ignore the baud rate change request. Fortunately, this routine supports all the common bps rates; if you need some other value, it is easy to modify this code to allow those other rates.

This code consists of two parts. The first part compares the value in ax against the set of valid bps values. If it finds a match, it loads ax with the corresponding 16 bit divisor constant. The second part of this code switches on the baud rate divisor registers and stores the value in ax into these registers. Finally, it switches the first two 8250 I/O registers back to the data and interrupt enable registers.

Note: This routine calls a few routines, notably ComSetLCR and ComGetLCR, that we will define a little later. These routines do the obvious functions, they read and write the LCR register (preserving registers, as appropriate).

```
ComBaud         proc
                push        ax
                push        dx
                cmp         ax, 9600
                ja          Set19200
                je          Set9600
                cmp         ax, 2400
                ja          Set4800
                je          Set2400
                cmp         ax, 600
                ja          Set1200
                je          Set600
                cmp         ax, 150
                ja          Set300
                je          Set150
                mov         ax, 1047        ;Default to 110 bps.
                jmp         SetPort

Set150:         mov         ax, 768         ;Divisor value for 150 bps.
                jmp         SetPort

Set300:         mov         ax, 384         ;Divisor value for 300 bps.
                jmp         SetPort

Set600:         mov         ax, 192         ;Divisor value for 600 bps.
                jmp         SetPort

Set1200:        mov         ax, 96          ;Divisor value for 1200 bps.
                jmp         SetPort

Set2400:        mov         ax, 48          ;Divisor value for 2400 bps.
                jmp         SetPort

Set4800:        mov         ax, 24          ;Divisor value for 4800 bps.
                jmp         SetPort

Set9600:        mov         ax, 12          ;Divisor value for 9600 bps.
                jmp         short SetPort
```

```
Set19200:       mov         ax, 6               ;Divisor value for 19.2 kbps.
SetPort:        mov         dx, ax              ;Save baud value.
                call        GetLCRCom           ;Fetch LCR value.
                push        ax                  ;Save old divisor bit value.
                or          al, 80h             ;Set divisor select bit.
                call        SetLCRCom           ;Write LCR value back.
                mov         ax, dx              ;Get baud rate divisor value.
                mov         dx, ComPort         ;Point at L.O. byte of divisor reg.
                out         dx, al              ;Output L.O. byte of divisor.
                inc         dx                  ;Point at the H.O. byte.
                mov         al, ah              ;Put H.O. byte in AL.
                out         dx, al              ;Output H.O. byte of divisor.
                pop         ax                  ;Retrieve old LCR value.
                call        SetLCRCom1          ;Restore divisor bit value.
                pop         dx
                pop         ax
                ret
ComBaud         endp
```

The ComStop routine programs the LCR to provide the specified number of stop bits. On entry, ax should contain either one or two (the number of stop bits you desire). This code converts that to zero or one and writes the resulting L.O. bit to the stop bit field of the LCR. Note that this code ignores the other bits in the ax register. This code reads the LCR, masks out the stop bit field, and then inserts the value the caller specifies into that field. Note the usage of the shl ax, 2 instruction; this requires an 80286 or later processor.

```
comStop         proc
                push        ax
                push        dx
                dec         ax                  ;Convert 1 or 2 to 0 or 1.
                and         al, 1               ;Strip other bits.
                shl         ax, 2               ;position into bit #2.
                mov         ah, al              ;Save our output value.
                call        ComGetLCR           ;Read LCR value.
                and         al, 11111011b       ;Mask out Stop Bits bit.
                or          al, ah              ;Merge in new # of stop bits.
                call        ComSetLCR           ;Write result back to LCR.
                pop         dx
                pop         ax
                ret
comStop         endp
```

The ComSize routine sets the word size for data transmission. As usual, this code provides a "programmer friendly" interface to the 8250 SCC. On enter, you specify the number of bits (5, 6, 7, or 8) in the ax register, you do not have to worry an appropriate bit pattern for the 8250's LCR register. This routine will compute the appropriate bit pattern for you. If the value in the ax register is not appropriate, this code defaults to an eight bit word size.

```
ComSize         proc
                push        ax
                push        dx
                sub         al, 5               ;Map 5..8 -> 00b, 01b, 10b, 11b
                cmp         al, 3
                jbe         Okay
                mov         al, 3               ;Default to eight bits.
Okay:           mov         ah, al              ;Save new bit size.
                call        ComGetLCR           ;Read current LCR value.
                and         al, 11111100b       ;Mask out old word size.
                or          al, ah              ;Merge in new word size.
                call        ComSetLCR           ;Write new LCR value back.
                pop         dx
                pop         ax
                ret
comsize         endp
```

The ComParity routine initializes the parity options on the 8250. Unfortunately, there is little possibility of a "programmer friendly" interface to this routine, So this code requires that you pass one of the following values in the ax register:

Table 86: ComParity Input Parameters

Value in AX	Description
0	Disable parity.
1	Enable odd parity checking.
3	Enable even parity checking.
5	Enable stuck parity bit with value one.
7	Enable stuck parity bit with value zero.

```
comparity       proc
                push        ax
                push        dx

                shl         al, 3               ;Move to final position in LCR.
                and         al, 00111000b       ;Mask out other data.
                mov         ah, al              ;Save for later.
                call        ComGetLCR           ;Get current LCR value.
                and         al, 11000111b       ;Mask out existing parity bits.
                or          al, ah              ;Merge in new bits.
                call        ComSetLCR           ;Write results back to the LCR.
                pop         dx
                pop         ax
                ret
comparity       endp
```

The next set of serial communication routines provide polled I/O support. These routines let you easily read characters from the serial port, write characters to the serial port, and check to see if there is data available at the input port or see if it is okay to write data to the output port. *Under no circumstances should you use these routines when you've activated the serial interrupt system.* Doing so may confuse the system and produce incorrect data or loss of data.

The ComRead routine is comparable to getc – it waits until data is available at the serial port, reads that data, and returns it in the al register. This routine begins by making sure we can access the Receive Data register (by clearing the baud rate divisor latch bit in the LCR).

```
ComRead         proc
                push        dx
                call        GetLCRCom
                push        ax                  ;Save divisor latch access bit.
                and         al, 7fh             ;Select normal ports.
                call        SetLCRCom           ;Write LCR to turn off divisor reg.
WaitForChar:    call        GetLSRCom           ;Get data available bit from LSR.
                test        al, 1               ;Data Available?
                jz          WaitForChar         ;Loop until data available.
                mov         dx, comPort         ;Read the data from the input port.
                in          al, dx
                mov         dl, al              ;Save character
                pop         ax                  ;Restore divisor access bit.
                call        SetLCRCom           ;Write it back to LCR.
                mov         al, dl              ;Restore output character.
                pop         dx
                ret
```

```
ComRead  endp
```

The ComWrite routine outputs the character in al to the serial port. It first waits until the transmitter holding register is empty, then it writes the output data to the output register.

```
ComWrite        proc
                push        dx
                push        ax
                mov         dl, al              ;Save character to output
                call        GetLCRCom           ;Switch to output register.
                push        ax                  ;Save divisor latch access bit.
                and         al, 7fh             ;Select normal input/output ports
                call        SetLCRCom           ; rather than divisor register.
WaitForXmtr:    call        GetLSRCom           ;Read LSR for xmit empty bit.
                test        al, 00100000b       ;Xmtr buffer empty?
                jz          WaitForXmtr         ;Loop until empty.
                mov         al, dl              ;Get output character.
                mov         dx, ComPort         ;Store it in the ouput port to
                out         dx, al              ; get it on its way.
                pop         ax                  ;Restore divisor access bit.
                call        SetLCRCom
                pop         ax
                pop         dx
                ret
ComWrite        endp
```

The ComTstIn and ComTstOut routines let you check to see if a character is available at the input port (ComTstIn) or if it is okay to send a character to the output port (ComTstOut). ComTstIn returns zero or one in al if data is not available or is available, respectively. ComTstOut returns zero or one in al if the transmitter register is full or empty, respectively.

```
ComTstIn        proc
                call        GetComLSR
                and         ax, 1               ;Keep only data available bit.
                ret
ComTstIn        endp


ComTstOut       proc
                push        dx
                call        ComGetLSR           ;Get the line status.
                test        al, 00100000b       ;Mask Xmitr empty bit.
                mov         al, 0               ;Assume not empty.
                jz          toc1                ;Branch if not empty.
                inc         ax                  ;Set to one if it is empty.
toc1:           ret
ComTstOut       endp
```

The next set of routines the Standard Library supplies load and store the various registers on the 8250 SCC. Although these are all trivial routines, they allow the programmer to access these register by name without having to know the address. Furthermore, these routines all preserve the value in the dx register, saving some code in the calling program if the dx register is already in use.

The following routines let you read ("Get") the value in the LSR, MSR, LCR, MCR, IIR, and IER registers, returning said value in the al register. They let you write ("Set") the value in al to any of the LCR, MCR, and IER registers. Since these routines are so simple and straight-forward, there is no need to discuss each routine individually. Note that you should avoid calling these routines outside an SCC ISR while in interrupt mode, since doing so can affect the interrupt system on the 8250 SCC.

```
ComGetLSR   proc                ;Returns the LSR value in the AL reg.
            push    dx
            mov     dx, comLSR  ;Select LSR register.
            in      al, dx      ;Read and return the LSR value.
            pop     dx
            ret
ComGetLSR   endp


ComGetMSR   proc                ;Returns the MSR value in the AL reg.
            push    dx
            mov     dx, comMSR  ;Select MSR register.
            in      al, dx      ;Read and return MSR value.
            pop     dx
            ret
ComGetMSR   endp


ComSetMCR   proc                ;Stores AL's value to the MCR reg.
            push    dx
            mov     dx, comMCR  ;Point at MCR register.
            out     dx, al      ;Output value in AL to MCR.
            pop     dx
            ret
ComSetMCR   endp


ComGetMCR   proc                ;Stores value in AL into MCR reg.
            push    dx
            mov     dx, comMCR  ;Select MCR register.
            in      al, dx      ;Read value from MCR register into AL.
            pop     dx
            ret
ComGetMCR   endp


ComGetLCR   proc                ;Return the LCR value in the AL reg.
            push    dx
            mov     dx, comLCR  ;Point at LCR register.
            in      al, dx      ;Read and return LCR value.
            pop     dx
            ret
ComGetLCR   endp


ComSetLCR   proc                ;Write a new value to the LCR.
            push    dx
            mov     dx, comLCR  ;Point at LCR register.
            out     dx, al      ;Write value in AL to the LCR.
            pop     dx
            ret
ComSetLCR   endp


ComGetIIR   proc                ;Return the value in the IIR.
            push    dx
            mov     dx, comIIR  ;Select IIR register.
            in      al, dx      ;Read IIR value into AL and return.
            pop     dx
            ret
ComGetIIR   endp
```

```
ComGetIER       proc                    ;Return IER value in AL.
                push        dx
                call        ComGetLCR   ;Need to select IER register by saving
                push        ax          ; the LCR value and then clearing the
                and         al, 7fh     ; baud rate divisor latch bit.
                call        ComSetLCR
                mov         dx, comIER  ;Address the IER.
                in          al, dx      ;Read current IER value.
                mov         dl, al      ;Save for now
                pop         ax          ;Retrieve old LCR value (divisor latch).
                call        ComSetLCR   ;Restore divisor latch
                mov         al, dl      ;Restore IER value
                pop         dx
                ret
ComGetIER       endp


ComSetIER       proc                    ;Writes value in AL to the IER.
                push        dx
                push        ax          ;Save AX's value.
                mov         ah, al      ;Save IER value to output.
                call        ComGetLCR   ;Get and save divsor access
                push        ax          ; bit.
                and         al, 7fh     ;Clear divisor access bit.
                call        ComSetLCR
                mov         al, ah      ;Retrieve new IER value.
                mov         dx, comIER  ;Select IER register
                out         dx, al      ;Output IER value.
                pop         ax          ;Restore divisor latch bit.
                call        ComSetLCR
                pop         ax
                pop         dx
                ret
ComSetIER       endp
```

The last set of serial support routines appearing in the Standard Library provide support for interrupt driven I/O. There are five routines in this section of the code: ComInitIntr, ComDisIntr, ComIntISR, ComIn, and ComOut. The ComInitIntr initializes the serial port interrupt system. It saves the old int 0Ch interrupt vector, initializes the vector to point at the ComIntISR interrupt service routine, and properly initializes the 8259A PIC and 8250 SCC for interrupt based operation. ComDisIntr undoes everything the ComDisIntr routine sets up; you need to call this routine to disable interrupts before your program quits. ComOut and ComIn transfer data to and from the buffers described in the variables section; the ComIntISR routine is responsible for removing data from the transmit queue and sending over the serial line as well as buffering up incoming data from the serial line.

The ComInitIntr routine initializes the 8250 SCC and 8259A PIC for interrupt based serial I/O. It also initializes the int 0Ch vector to point at the ComIntISR routine. One thing this code does *not* do is to provide break and critical error exception handlers. Remember, if the user hits ctrl-C (or ctrl-Break) or selects abort on an I/O error, the default exception handlers simply return to DOS without restoring the int 0Ch vector. It is important that your program provide exception handlers that will call ComDisIntr before allowing the system to return control to DOS. Otherwise the system may crash when DOS loads the next program into memory. See "Interrupts, Traps, and Exceptions" on page 995 for more details on writing these exception handlers.

```
ComInitIntr     proc
                pushf                   ;Save interrupt disable flag.
                push        es
                push        ax
                push        dx

; Turn off the interrupts while we're doing this.

                cli
```

```
; Save old interrupt vector. Obviously, you must change the following code
; to save and set up the int 0Bh vector if you want to access COM2: rather
; than the COM1: port.

                xor     ax, ax          ;Point at interrupt vectors
                mov     es, ax
                mov     ax, Int0Cofs
                mov     word ptr OldIInt0C, ax
                mov     ax, Int0Cseg
                mov     word ptr OldInt0C+2, ax

; Point int 0ch vector at our interrupt service routine (see note above
; concerning switching to COM2:).

                mov     ax, cs
                mov     Int0Cseg, ax
                mov     ax, offset ComIntISR
                mov     Int0Cofs, ax

; Clear any pending interrupts:

                call    ComGetLSR       ;Clear Receiver line status
                call    ComGetMSR       ;Clear CTS/DSR/RI Interrupts
                call    ComGetIIR       ;Clear xmtr empty interrupt
                mov     dx, ComPort
                in      al, dx          ;Clear data available intr.

; Clear divisor latch access bit. WHILE OPERATING IN INTERRUPT MODE, THE
; DIVISOR ACCESS LATCH BIT MUST ALWAYS BE ZERO. If for some horrible reason
; you need to change the baud rate in the middle of a transmission (or while
; the interrupts are enabled) clear the interrupt flag, do your dirty work,
; clear the divisor latch bit, and finally restore interrupts.

                call    ComGetLCR       ;Get LCR.
                and     al, 7fh         ;Clear divisor latch bit.
                call    ComSetLCR       ;Write new LCR value back.


; Enable the receiver and transmitter interrupts. Note that this code
; ignores error and modem status change interrupts.

                mov     al, 3         ;Enable rcv/xmit interrupts
                call    SetIERCom

; Must set the OUT2 line for interrupts to work.
; Also sets DTR and RTS active.

                mov     al, 00001011b
                call    ComSetMCR

; Activate the COM1 (int 0ch) bit in the 8259A interrupt controller chip.
; Note: you must change the following code to clear bit three (rather than
; four) to use this code with the COM2: port.

                in      al, 21h       ;Get 8259A interrupt enable value.
                mov     i8259a, al    ;Save interrupt enable bits.
                and     al, 0efh      ;Bit 4=IRQ 4 = INT 0Ch
                out     21h, al       ;Enable interrupts.

                pop     dx
                pop     ax
                pop     es
                popf                  ;Restore interrupt disable flag.
                ret
ComInitIntr     endp
```

The ComDisIntr routine disables serial interrupts. It restores the original value of the 8259A interrupt enable register, it restores the int 0Ch interrupt vector, and it masks interrupts on the 8250 SCC. Note that this code assumes that you have not changed the interrupt enable bits in the 8259 PIC since calling

ComInitIntr. It restores the 8259A's interrupt enable register with the value from the 8259A interrupt enable register when you originally called ComInitIntr.

It would be a complete disaster to call this routine without first calling ComInitIntr. Doing so would patch the int 0Ch vector with garbage and, likewise, restore the 8259A interrupt enable register with a garbage value. Make sure you've called ComInitIntr before calling this routine. Generally, you should call ComInitIntr once, at the beginning of your program, and call ComDisIntr once, either at the end of your program or within the break or critical error exception routines.

```
ComDisIntr      proc
                pushf
                push        es
                push        dx
                push        ax

                cli                     ;Don't allow interrupts while messing
                xor         ax, ax      ; with the interrupt vectors.
                mov         es, ax      ;Point ES at interrupt vector table.

; First, turn off the interrupt source at the 8250 chip:

                call        ComGetMCR   ;Get the OUT 2 (interrupt enable) bit.
                and         al, 3       ;Mask out OUT 2 bit (masks ints)
                call        ComSetMCR   ;Write result to MCR.

; Now restore the IRQ 4 bit in the 8259A PIC. Note that you must modify this
; code to restore the IRQ 3 bit if you want to support COM2: instead of COM1:

                in          al, 21h     ;Get current 8259a IER value
                and         al, 0efh    ;Clear IRQ 4 bit (change for COM2:!)
                mov         ah, i8259a  ;Get our saved value
                and         ah, 1000b   ;Mask out com1: bit (IRQ 4).
                or          al, ah      ;Put bit back in.
                out         21h, al

; Restore the interrupt vector:

                mov         ax, word ptr OldInt0C
                mov         Int0Cofs, ax
                mov         ax, word ptr OldInt0C+2
                mov         Int0Cseg, ax

                pop         ax
                pop         dx
                pop         es
                popf
                ret
ComDisIntr      endp
```

The following code implements the interrupt service routine for the 8250 SCC. When an interrupt occurs, this code reads the 8250 IIR to determine the source of the interrupt. The Standard Library routines only provide direct support for data available interrupts and transmitter holding register empty interrupts. If this code detects an error or status change interrupt, it clears the interrupt status but takes no other action. If it detects a receive or transmit interrupt, it transfers control to the appropriate handler.

The receiver interrupt handler is very easy to implement. All this code needs to do is read the character from the Receive Register and add this character to the input buffer. The only catch is that this code must ignore any incoming characters if the input buffer is full. An application can access this data using the ComIn routine that removes data from the input buffer.

The transmit handler is somewhat more complex. The 8250 SCC interrupts the 80x86 when it is able to accept more data for transmission. However, the fact that the 8250 is ready for more data doesn't guarantee there is data ready for transmission. The application produces data at its own rate, not necessarily at the rate that 8250 SCC wants it. Therefore, it is quite possible for the 8250 to say "give me more data" but

the application has not produced any. Obviously, we should not transmit anything at that point. Instead, we have to wait for the application to produce more data before transmission resumes.

Unfortunately, this complicates the driver for the transmission code somewhat. With the receiver, the interrupt always indicates that the ISR can move data from the 8250 to the buffer. The application can remove this data at any time and the process is always the same: wait for a non-empty receive buffer and then remove the first item from the buffer. Unfortunately, we cannot simply do the converse operation when transmitting data. That is, we can't simply store data in the transmit buffer and leave it up to the ISR to remove this data. The problem is that the 8250 only interrupts the system once when the transmitter holding register is empty. If there is no data to transmit at that point, the ISR must return without writing anything to the transmit register. *Since there is no data in the transmit buffer, there will be no additional transmitter interrupts generated, even when there is data added to the transmit buffer*. Therefore, the ISR and the routine responsible for adding data to the output buffer (ComOut) must coordinate their activities. If the buffer is empty and the transmitter is not currently transmitting anything, the ComOut routine must write its data directly to the 8250. If the 8250 is currently transmitting data, ComOut must append its data to the end of the output buffer. The ComIntISR and ComOut use a flag, TestBuffer, to determine whether ComOut should write directly to the serial port or append its data to the output buffer. See the following code and the code for ComOut for all the details.

```
ComIntISR       proc        far
                push        ax
                push        bx
                push        dx
TryAnother:     mov         dx, ComIIR
                in          al, dx              ;Get interrupt id value.
                test        al, 1               ;Any interrupts left?
                jnz         IntRtn              ;Quit if no interrupt pending.
                cmp         al, 100b            ;Since only xmit/rcv ints are
                jnz         ReadCom1            ; active, this checks for rcv int.
                cmp         al, 10b             ;This checks for xmit empty intr.
                jnz         WriteCom1

; Bogus interrupt? We shouldn't ever fall into this code because we have
; not enabled the error or status change interrupts. However, it is possible
; that the application code has gone in and tweakd the IER on the 8250.
; Therefore, we need to supply a default interrupt handler for these conditions.
; The following code just reads all the appropriate registers to clear any
; pending interrupts.

                call        ComGetLSR           ;Clear receiver line status
                call        ComGetMSR           ;Clear modem status.
                jmp         TryAnother          ;Check for lower priority intr.

; When there are no more pending interrupts on the 8250, drop down and
; and return from this ISR.

IntRtn:         mov         al, 20h             ;Acknowledge interrupt to the
                out         20h, al             ; 8259A interrupt controller.
                pop         dx
                pop         bx
                pop         ax
                iret

; Handle incoming data here:
; (Warning: This is a critical region. Interrupts MUST BE OFF while executing
; this code. By default, interrupts are off in an ISR. DO NOT TURN THEM ON
; if you modify this code).

ReadCom1:       mov         dx, ComPort         ;Point at data input register.
                in          al, dx              ;Get the input char

                mov         bx, InHead          ;Insert the character into the
                mov         [bx], al            ; serial input buffer.

                inc         bx                  ;Increment buffer ptr.
                cmp         bx, offset InpBufEnd
                jb          NoInpWrap
```

```
                mov     bx, offset InpBuf
NoInpWrap:      cmp     bx, InTail      ;If the buffer is full, ignore this
                je      TryAnother      ; input character.
                mov     InHead, bx
                jmp     TryAnother      ;Go handle other 8250 interrupts.


; Handle outgoing data here (This is also a critical region):

WriteCom1:      mov     bx, OutTail     ;See if the buffer is empty.
                cmp     bx, OutHead
                jne     OutputChar      ;If not, output the next char.

; If head and tail are equal, simply set the TestBuffer variable to zero
; and quit. If they are not equal, then there is data in the buffer and
; we should output the next character.

                mov     TestBuffer, 0
                jmp     TryAnother      ;Handle other pending interrupts.

; The buffer pointers are not equal, output the next character down here.

OutputChar:     mov     al, [bx]        ;Get the next char from the buffer.
                mov     dx, ComPort     ;Select output port.
                out     dx, al          ;Output the character

; Okay, bump the output pointer.

                inc     bx
                cmp     bx, offset OutBufEnd
                jb      NoOutWrap
                mov     bx, offset OutBuf
NoOutWrap:      mov     OutTail, bx
                jmp     TryAnother
ComIntISR       endp
```

These last two routines read data from the serial input buffer and write data to the serial output buffer. The ComIn routine, that handles the input chore, waits until the input buffer is not empty. Then it removes the first available byte from the input buffer and returns this value to the caller.

```
ComIn           proc
                pushf                   ;Save interrupt flag
                push    bx
                sti                     ;Make sure interrupts are on.
TstInLoop:      mov     bx, InTail      ;Wait until there is at least one
                cmp     bx, InHead      ; character in the input buffer.
                je      TstInLoop
                mov     al, [bx]        ;Get next char.
                cli                     ;Turn off ints while adjusting
                inc     bx              ; buffer pointers.
                cmp     bx, offset InpBufEnd
                jne     NoWrap2
                mov     bx, offset InpBuf
NoWrap2:        mov     InTail, bx
                pop     bx
                popf                    ;Restore interrupt flag.
                ret
ComIn           endp
```

The ComOut must check the TestBuffer variable to see if the 8250 is currently busy. If not (TestBuffer equals zero) then this code must write the character directly to the serial port and set TestBuffer to one (since the chip is now busy). If the TestBuffer contains a non-zero value, this code simply appends the character in al to the end of the output buffer.

```
ComOut          proc        far
                pushf
                cli                             ;No interrupts now!
                cmp         TestBuffer, 0       ;Write directly to serial chip?
                jnz         BufferItUp          ;If not, go put it in the buffer.

; The following code writes the current character directly to the serial port
; because the 8250 is not transmitting anything now and we will never again
; get a transmit holding register empty interrupt (at least, not until we
; write data directly to the port).

                push        dx
                mov         dx, ComPort         ;Select output register.
                out         dx, al              ;Write character to port.
                mov         TestBuffer, 1       ;Must buffer up next char.
                pop         dx
                popf                            ;Restore interrupt flag.
                ret

; If the 8250 is busy, buffer up the character here:

BufferItUp:     push        bx
                mov         bx, OutHead         ;Pointer to next buffer position.
                mov         [bx], al            ;Add the char to the buffer.

; Bump the output pointer.

                inc         bx
                cmp         bx, offset OutBufEnd
                jne         NoWrap3
                mov         bx, offset OutBuf
NoWrap3:        cmp         bx, OutTail         ;See if the buffer is full.
                je          NoSetTail           ;Don't add char if buffer is full.
                mov         OutHead, bx         ;Else, update buffer ptr.
NoSetTail:      pop         bx
                popf                            ;Restore interrupt flag
                ret
ComOut          endp
```

Note that the Standard Library does not provide any routines to see if there is data available in the input buffer or to see if the output buffer is full (comparable to the ComTstIn and ComTstOut routines). However, these are very easy routines to write; all you need do is compare the head and tail pointers of the two buffers. The buffers are empty if the head and tail pointers are equal. The buffers are full if the head pointer is one byte before the tail pointer (keep in mind, the pointers wrap around at the end of the buffer, so the buffer is also full if the head pointer is at the last position in the buffer and the tail pointer is at the first position in the buffer).

22.4 Summary

This chapter discusses RS-232C serial communications on the PC. Like the parallel port, there are three levels at which you can access the serial port: through DOS, through BIOS, or by programming the hardware directly. Unlike DOS' and BIOS' parallel printer support, the DOS serial support is almost worthless and the BIOS support is rather weak (e.g., it doesn't support interrupt driven I/O). Therefore, it is common programming practice on the PC to control the hardware directly from an application program. Therefore, familiarizing one's self with the 8250 Serial Communication Chip (SCC) is important if you intend to do serial communications on the PC. This chapter does not discuss serial communication from DOS or BIOS, mainly because their support is so limited. For further information on programming the serial port from DOS or BIOS, see "MS-DOS, PC-BIOS, and File I/O" on page 699.

The 8250 supports ten I/O registers that let you control the communication parameters, check the status of the chip, control interrupt capabilities, and, of course, perform serial I/O. the 8250 maps these registers to eight I/O locations in the PC's I/O address space.

The PC supports up to four serial communication devices: COM1:, COM2:, COM3:, and COM4:. However, most software only deals with the COM1: and COM2: ports. Like the parallel port support, BIOS differentiates logical communication ports and physical communication ports. BIOS stores the base address of COM1:..COM4: in memory locations 40:0, 40:2, 40:4, and 40:6. This base address is the I/O address of the first 8250 register for that particular communication port. For more information on the 8250 hardware, check out

- "The 8250 Serial Communications Chip" on page 1223
- "The Data Register (Transmit/Receive Register)" on page 1224
- "The Interrupt Enable Register (IER)" on page 1224
- "The Baud Rate Divisor" on page 1225
- "The Interrupt Identification Register (IIR)" on page 1226
- "The Line Control Register" on page 1227
- "The Modem Control Register" on page 1228
- "The Line Status Register (LSR)" on page 1229
- "The Modem Status Register (MSR)" on page 1230
- "The Auxiliary Input Register" on page 1231

The UCR Standard Library provides a very reasonable set of routines you can use to control the serial port on the PC. Not only does this package provide a set of polling routines you can use much like the BIOS' code, but it also provides an interrupt service routine to support interrupt driven I/O on the serial port. For more information on these routines, see

- "The UCR Standard Library Serial Communications Support Routines" on page 1231

The Standard Library serial I/O routines provide an excellent example of how to program the 8250 SCC. Therefore, this chapter concludes by presenting and explaining the Standard Library's serial I/O routines. In particular, this code demonstrates some of the subtle problems with interrupt driven serial communication. For all the details, read

- "Programming the 8250 (Examples from the Standard Library)" on page 1233

The PC Video Display Chapter 23

The PC's video display is a very complex system. First, there is not a single common device as exists for the parallel and serial ports, or even a few devices (like the keyboard systems found on PCs). No, there are literally dozens of different display adapter cards available for the PC. Furthermore, each adapter typically supports several different display modes. Given the large number of display modes and uses for the display adapters, it would be very easy to write a book as large as this one on the PC's display adapters alone[1] However, this is not that text. This book would hardly be complete without at least mentioning the PC's video display, but there are not enough pages remaining in this book to do justice to the subject. Therefore, this chapter will discuss the 80 x 25 *text display mode* that nearly all display adapters support.

23.1 Memory Mapped Video

Most peripheral devices on the PC use *I/O mapped* input/output. A program communicates with I/O mapped devices using the 80x86 `in`, `out`, `ins`, and `outs` instructions, accessing devices in the PC's I/O address space. While the video controller chips that appear on PC video display adapters also map registers to the PC's I/O space, these cards also employ a second form of I/O addressing: *memory mapped I/O* input/output. In particular, the 80 x 25 text display is nothing more than a two dimensional array of words with each word in the array corresponding a character on the screen. This array appears just above the 640K point in the PC's memory address space. If you store data into this array using standard memory addressing instruction (e.g., `mov`), you will affect the characters appearing on the display.

There are actually two different arrays you need to worry about. Monochrome system (remember those?) locate their text display starting at location B000:0000 in memory. Color systems locate their text displays at location B800:0000 in memory. These locations are the base addresses of a column major order array declared as follows:

```
Display: array [0..79, 0..24] of word;
```

If you prefer to work with row major ordered arrays, no problem, the video display is equal to the following array definition:

```
Display: array [0..24, 0..79] of word;
```

Note that location (0,0) is the upper left hand corner and location (79,24) is the lower right hand corner of the display (the values in parentheses are the x and y coordinates, with the x/horizontal coordinate appearing first).

The L.O. byte of each word contains the PC/ASCII code for the character you want to display (see Appendix A for a listing of the PC/ASCII character set). The H.O. byte of each word is the *attribute byte*. We will return to the attribute byte in the next section.

The display page consumes slightly less than 4 Kilobytes in the memory map. The color display adapters actually provide 32K for text displays and let you select one of eight different displays. Each such display begins on a 4K boundary, at address B800:0, B800:1000, B800:2000, B800:3000, ..., B800:7000. Note that most modern color display adapters actually provide memory from address A000:0 through B000:FFFF (and more), but the text display only uses the 32K from B800:0..B800:7FFF. In this chapter, we will only concern ourselves with the first color display page at address B800:0. However, everything discussed in this chapter applies to the other display pages as well.

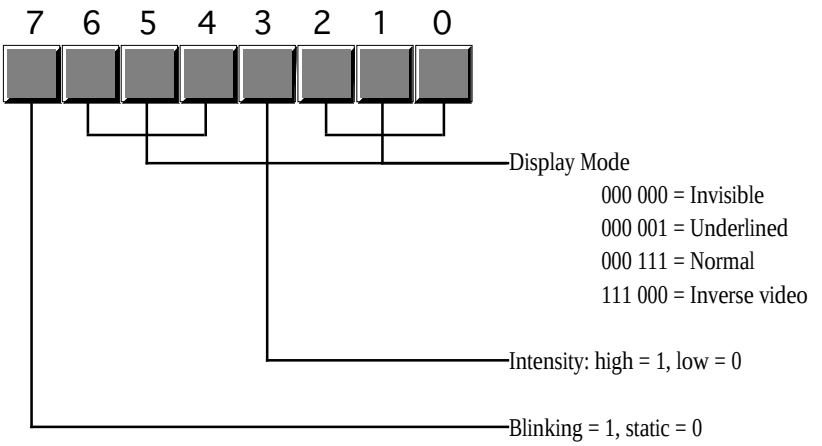
The monochrome adapter provides only a single display page. Indeed, the earliest monochrome display adapters included only 4K on-board memory (contrast this with modern high density color display adapters that have up to four megabytes of on-board memory!).

1. In fact, several such books exist. See the bibliography.

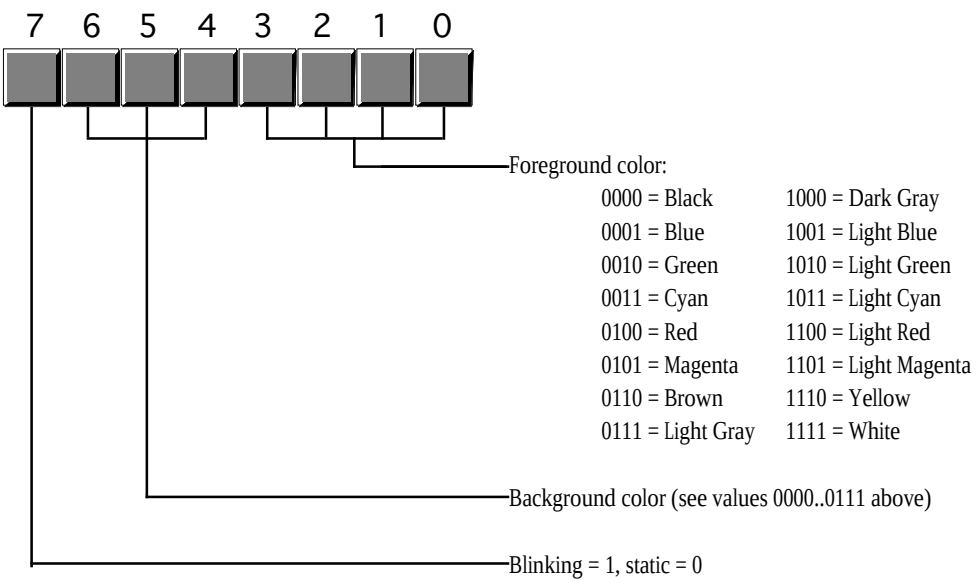
You can address the memory on the video display like ordinary RAM. You could even store program variables, or even code, in this memory. However, it is never a good idea to do this. First of all, any time you write to the display screen, you will wipe out any variables stored in the active display memory. Even if you store such code or variables in an inactive display page (e.g., pages one through seven on a color display adapter), using this memory in this manner is not a good idea because access to the display adapter is very slow. Main memory runs two to ten times faster (depending on the machine).

23.2 The Video Attribute Byte

The video attribute associated with each character on the screen controls underlining, intensity, inverse video, and blinking video on monochrome adapters. It controls blinking and character foreground/background colors on color displays. The following diagrams provide the possible attribute values:



Monochrome Display Adapter Attribute Byte Format



Color Display Adapter Attribute Byte Format

To get reverse video on the color display, simply swap the foreground and background colors. Note that a foreground color of zero with a background color of seven produces black characters on a white background, the standard reverse video colors and the same attribute values you'd use on the monochrome display adapter.

You need to be careful when choosing foreground and background colors for text on a color display adapters. Some combinations are impossible to read (e.g., white characters on a white background). Other colors go together so poorly the text will be extremely difficult to read, if not impossible (how about light green letters on a green background?). You must choose your colors with care!

Blinking characters are great for drawing attention to some important text on the screen (like a warning). However, it is easy to overdo blinking text on the screen. You should never have more than one word or phrase blinking on the screen at one time. Furthermore, you should never leave blinking characters on the screen for any length of time. After a few seconds, replace blinking characters with normal characters to avoid annoying the user of your software.

Keep in mind, you can easily change the attributes of various characters on the screen without affecting the actual text. Remember, the attribute bytes appear at odd addresses in the memory space for the video display. You can easily go in and change these bytes while leaving the character code data alone.

23.3 Programming the Text Display

You might ask why anyone would want to bother working directly with the memory mapped display on the PC. After all, DOS, BIOS, and the UCR Standard Library provide *much* more convenient ways to display text on the screen. Handling new lines (carriage return and line feed) at the end of each line or, worse yet, scrolling the screen when the display is full, is a lot of work. Work that is taken care of for you automatically by the aforementioned routines. Work you have to do yourself if you access screen memory directly. So why bother?

There are two reasons: performance and flexibility. The BIOS video display routines[2] are *dreadfully* slow. You can easily get a 10 to 100 times performance boost by writing directly to screen memory. For a typical computer science class project, this may not be important, especially if you're running on a fast machine like a 150 MHz Pentium. On the other hand, if you are developing a program that displays and removes several windows or pop-up menus on the screen, the BIOS routines won't cut it.

Although the BIOS int 10h functions provide a large set of video I/O routines, there will be lots of functions you might want to have that the BIOS just doesn't provide. In such cases, going directly to screen memory is one way to solve the problem.

Another difficulty with BIOS routine is that they are not reentrant. You cannot call a BIOS display function from an interrupt service routine, nor can you freely call BIOS from concurrently executing processes. However, by writing your own video service routines, you can easily create a window for each concurrent thread you application is executing. Then each thread can call your routines to display its output independent of the other threads executing on the system.

The AMAZE.ASM program (see "Processes, Coroutines, and Concurrency" on page 1065) is a good example of a program that directly access the text display by directly storing data into the video display's memory mapped display array. This program access display memory directly because it is more convenient to do so (the screen's display array maps quite nicely to the internal maze array). Simple video games like a space invaders game or a "remove the bricks" game also map nicely to a memory mapped video display.

The following program provides an excellent example of an application that needs to access video memory directly. This program is a *screen capture* TSR. When you press the left shift key and then the right shift key, this program copies the current screen contents to an internal buffer. When you press the

2. The Standard Library calls DOS and DOS calls BIOS for all display I/O, hence they all become BIOS calls at one level or another.

right shift key followed by the left shift key, this program copies its internal buffer to the display screen. Originally, this program was written to capture CodeView screens for the lab manual accompanying this text. There are commercial screen capture programs (e.g., HiJak) that would normally do the job, but are incompatible with CodeView. This short TSR allows one to capture screens in CodeView, quit CodeView, put the CodeView screen back onto the display, and the use a program like HiJak to capture the output.

```
; GRABSCRN.ASM
;
; A short TSR to capture the current display screen and display it later.
;
; Note that this code does not patch into int 2Fh (multiplex interrupt)
; nor can you remove this code from memory except by rebooting.
; If you want to be able to do these two things (as well as check for
; a previous installation), see the chapter on resident programs. Such
; code was omitted from this program because of length constraints.
;
;
; cseg and EndResident must occur before the standard library segments!

cseg            segment    para public 'code'
OldInt9         dword      ?
ScreenSave      byte       4096 dup (?)
cseg            ends

; Marker segment, to find the end of the resident section.

EndResident     segment    para public 'Resident'
EndResident     ends

                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list


RShiftScan      equ        36h
LShiftScan      equ        2ah

; Bits for the shift/modifier keys

RShfBit         equ        1
LShfBit         equ        2

KbdFlags        equ        <byte ptr ds:[17h]>

byp             equ        <byte ptr>


; Screen segment address. This value is for color displays only.
; Change to B000h if you want to use this program on a mono display.

ScreenSeg       equ        0B800h

cseg            segment    para public 'code'
                assume     ds:nothing



; MyInt9-       INT 9 ISR. This routine reads the keyboard port to see
;               if a shift key scan code just came along. If the right
;               shift bit is set in KbdFlags the a left shift key scan
;               code comes along, we want to copy the data from our
;               internal buffer to the screen's memory. If the left shift
;               bit is set and a right shift key scan code comes along,
;               we want to copy the screen memory into our local array.
;               In any case (including none of the above), we always transfer
;               control to the original INT 9 handler.

MyInt9          proc       far
                push       ds
                push       ax
```

```
                mov     ax, 40h
                mov     ds, ax

                in      al, 60h         ;Read the keyboard port.
                cmp     al, RShiftScan  ;Right shift just go down?
                je      DoRight
                cmp     al, LShiftScan  ;How about the left shift?
                jne     QuitMyInt9

; If this is the left scan code, see if the right shift key is already
; down.

                test    KbdFlags, RShfBit
                je      QuitMyInt9      ;Branch if no

; Okay, right shift was down and we just saw left shift, copy our local
; data back to screen memory:

                pushf
                push    es
                push    cx
                push    di
                push    si
                mov     cx, 2048
                mov     si, cs
                mov     ds, si
                lea     si, ScreenSave
                mov     di, ScreenSeg
                mov     es, di
                xor     di, di
                jmp     DoMove

; Okay, we just saw the right shift key scan code, see if the left shift
; key is already down. If so, save the current screen data to our local
; array.

DoRight:        test    KbdFlags, LShfBit
                je      QuitMyInt9

                pushf
                push    es
                push    cx
                push    di
                push    si
                mov     cx, 2048
                mov     ax, cs
                mov     es, ax
                lea     di, ScreenSave
                mov     si, ScreenSeg
                mov     ds, si
                xor     si, si

DoMove:         cld
        rep     movsw
                pop     si
                pop     di
                pop     cx
                pop     es
                popf
QuitMyInt9:
                pop     ax
                pop     ds
                jmp     OldInt9
MyInt9          endp



Main            proc
                assume  ds:cseg

                mov     ax, cseg
                mov     ds, ax

                print
```

```
                byte    "Screen capture TSR",cr,lf
                byte    "Pressing left shift, then right shift, captures "
                byte    "the current screen.",cr,lf
                byte    "Pressing right shift, then left shift, displays "
                byte    "the last captured screen.",cr,lf
                byte    0

; Patch into the INT 9 interrupt vector. Note that the
; statements above have made cseg the current data segment,
; so we can store the old INT 9 value directly into
; the OldInt9 variable.

                cli                     ;Turn off interrupts!
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[9*4]
                mov     word ptr OldInt9, ax
                mov     ax, es:[9*4 + 2]
                mov     word ptr OldInt9+2, ax
                mov     es:[9*4], offset MyInt9
                mov     es:[9*4+2], cs
                sti                     ;Okay, ints back on.


; We're hooked up, the only thing that remains is to terminate and
; stay resident.

                print
                byte    "Installed.",cr,lf,0

                mov     ah, 62h         ;Get this program's PSP
                int     21h             ; value.

                mov     dx, EndResident ;Compute size of program.
                sub     dx, bx
                mov     ax, 3100h     ;DOS TSR command.
                int     21h
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             db      1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

23.4 Summary

The PC's video system uses a memory mapped array for the screen data. This is an 80 x 25 column major organized array of words. Each word in the array corresponds to a single character on the screen. This array begins at location B000:0 for monochrome displays and B800:0 for color displays. For additional information, see:

- "Memory Mapped Video" on page 1247

The L.O. byte is the PC/ASCII character code for that particular screen position, the H.O. byte contains the attributes for that character. The attribute selects blinking, intensity, and background/foreground colors (on a color display). For more information on the attribute byte, see:

- "The Video Attribute Byte" on page 1248

There are a few reasons why you would want to bother accessing display memory directly. Speed and flexibility are the two primary reasons people go directly to screen memory. You can create your own

screen functions that the BIOS doesn't support and do it one or two orders of magnitude faster than the BIOS by writing directly to screen memory. To find out about this, and to see a simple example, check out

- "Programming the Text Display" on page 1249

One need look no farther than the internals of several popular games on the PC to discover than many programmers do not fully understand one of the least complex devices attached to the PC today – the analog game adapter. This device allows a user to connect up to four resistive potentiometers and four digital switch connections to the PC. The design of the PC's game adapter was obviously influenced by the analog input capabilities of the Apple II computer[1], the most popular computer available at the time the PC was developed. Although IBM provided for twice the analog inputs of the Apple II, thinking that would give them an edge, their decision to support only four switches and four potentiometers (or "pots") seems confining to game designers today – in much the same way that IBM's decision to support 256K RAM seems so limiting today. Nevertheless, game designers have managed to create some really marvelous products, even living with the limitations of IBM's 1981 design.

IBM's analog input design, like Apple's, was designed to be dirt cheap. Accuracy and performance were not a concern at all. In fact, you can purchase the electronic parts to build your own version of the game adapter, at retail, for under three dollars. Indeed, today you can purchase a game adapter card from various discount merchants for under eight dollars. Unfortunately, IBM's low-cost design in 1981 produces some major performance problems for high-speed machines and high-performance game software in the 1990's. However, there is no use crying over spilled milk – we're stuck with the original game adapter design, we need to make the most of it. The following sections will describe how to do exactly that.

24.1 Typical Game Devices

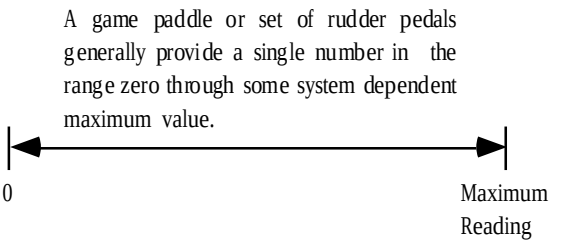
The game adapter is nothing more than a computer interface to various game input devices. The game adapter card typically contains a DB15 connector into which you plug an external device. Typical devices you can obtain for the game adapter include *paddles, joysticks, flight yokes, digital joysticks, rudder pedals, RC simulators,* and *steering wheels.* Undoubtedly, this is but a short list of the types of devices you can connect to the game adapter. Most of these devices are far more expensive that the game adapter card itself. Indeed, certain high performance flight simulator consoles for the game adapter cost several hundred dollars.

The digital joystick is probably the least complex device you can connect to the PC's game port. This device consists of four switches and a stick. Pushing the stick forward, left, right, or pulling it backward closes one of the switches. The game adapter card provides four switch inputs, so you can sense which direction (including the rest position) the user is pressing the digital joystick. Most digital joysticks also allow you to sense the in-between positions by closing two contacts at once. for example, pushing the control stick at a 45 degree angle between forward and right closes both the forward and right switches. The application software can sense this and take appropriate action. The original allure of these devices is that they were very cheap to manufacture (these were the original joysticks found on most home game machines). However, as manufacturers increased production of analog joysticks, the price fell to the point that digital joysticks failed to offer a substantial price difference. So today, you will rarely encounter such devices in the hands of a typical user.

The game paddle is another device whose use has declined over the years. A game paddle is a single pot in a case with a single knob (and, typically, a single push button). Apple used to ship a pair of game paddles with every Apple II they sold. As a result, games that used game paddles were still quite popular when IBM released the PC in 1981. Indeed, a couple manufacturers produced game paddles for the PC when it was first introduced. However, once again the cost of manufacturing analog joysticks fell to the point that paddles couldn't compete. Although paddles are the appropriate input device for many games, joysticks could do just about everything a game paddle could, and more. So the use of game paddles quickly died out. There is one thing you can do with game paddles that you cannot do with joysticks – you

1. In fact, the PC's game adapter design was obviously stolen directly from the Apple II.

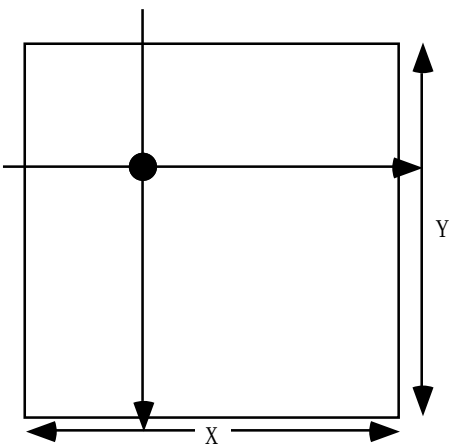
can place four of them on a system and produce a four player game. However, this (obviously) isn't important to most game designers who generally design their games for only one player.



Game Paddle or Rudder Pedal Game Input Device

Rudder pedals are really nothing more than a specially designed game paddle designed so you can activate them with your feet. Many flight simulator games take advantage of this input device to provide a more realistic experience. Generally, you would use rudder pedals in addition to a joystick device.

A joystick contains two pots connected with a stick. Moving the joystick along the x-axis actuates one of the pots, moving the joystick along the y-axis actuates the other pot. By reading both pots, you can roughly determine the absolute position of the pot within its working range.



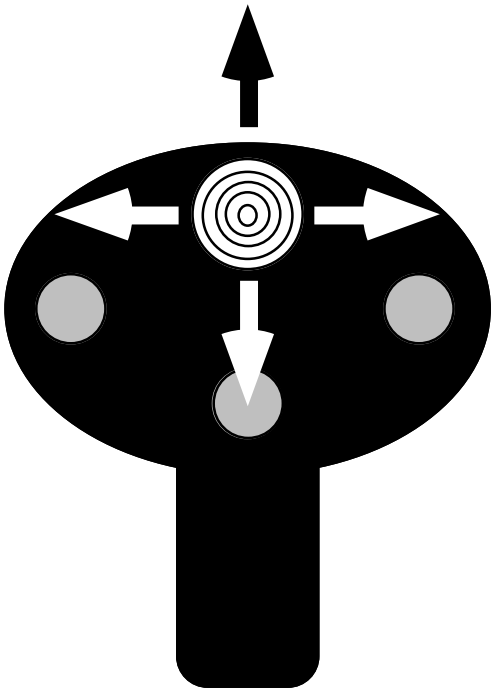
A joystick uses two independent pots to provide an (X,Y) input value. Horizontal movements on the joystick affect the x-axis pot independently of the y-axis pot. Likewise, vertical movements affect the y-axis pot independent of the x-axis pot. By reading both pots you can determine the position of the joystick in the (X,Y) coordinate system.

Joystick Game Input Device

An RC simulator is really nothing more than a box containing two joysticks. The yoke and steering wheel devices are essentially the same device, sold specifically for flight simulators or automotive games[2]. The steering wheel is connected to a pot that corresponds to the x-axis on the joystick. Pulling back (or pushing forward) on the wheel activates a second pot that corresponds to the y-axis on the joystick.

Certain joystick devices, generically known as *flight sticks*, contain three pots. Two pots are connected in a standard joystick fashion, the third is connected to a knob which many games use for the throttle control. Other joysticks, like the Thrustmaster™or CH Products' FlightStick Pro, include extra switches including a special "cooley switch" that provide additional inputs to the game. The cooley switch is, essentially, a digital pot mounted on the top of a joystick. Users can select one of four positions on the cooley switch using their thumb. Most flight simulator programs compatible with such devices use the cooley switch to select different views from the aircraft.

2. In fact, many such devices are switchable between the two.

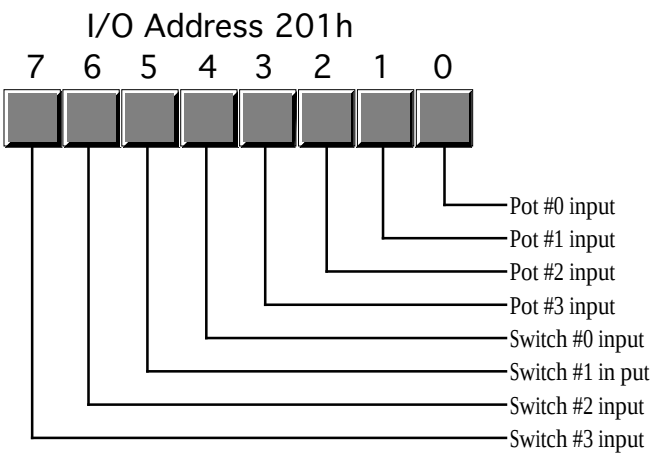


The cooley switch (shown here on a device layout similar to the CH Products' FlightStick Pro) is a thumb actuated digitial joystick. You can move the switch up, down, left or right, activating individual switches inside the game input device.

Cooley Switch (found on CH Products and Thrustmaster Joysticks)

24.2 The Game Adapter Hardware

The game adapter hardware is simplicity itself. There is a single input port and a single output port. The input port bit layout is

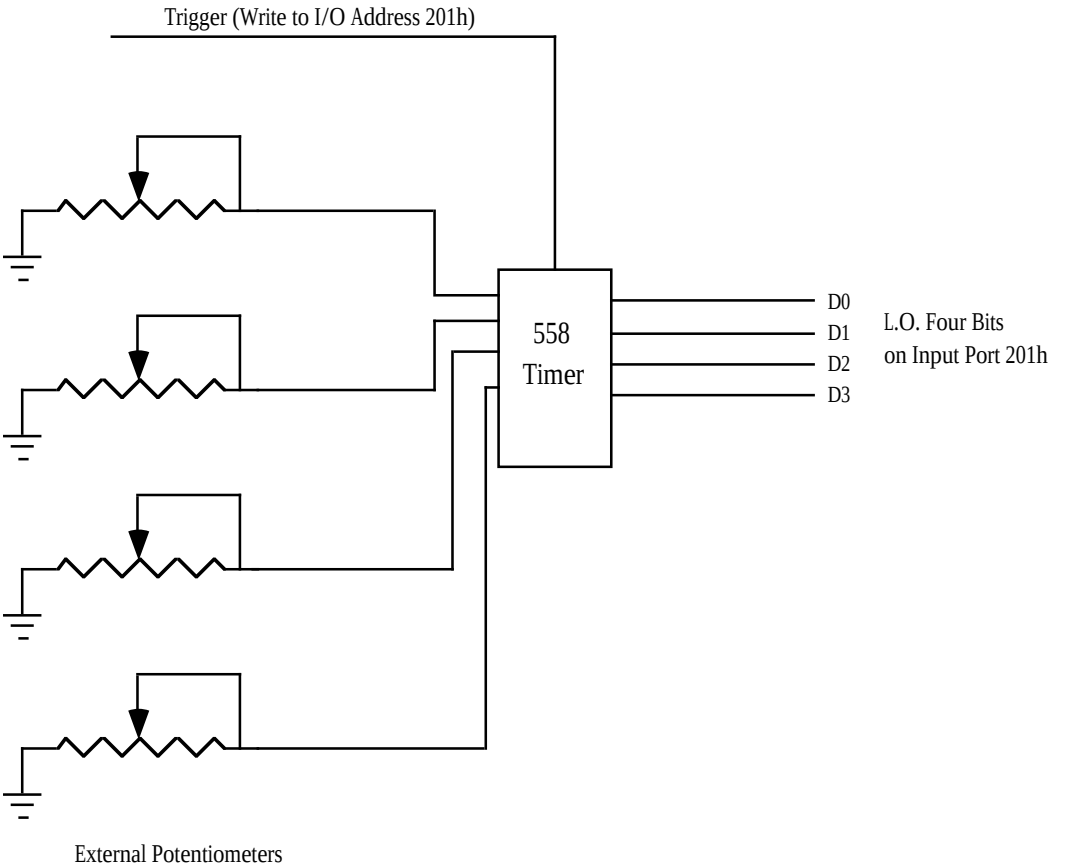


Game Adapter Input Port

The four switches come in on the H.O. four bits of I/O port 201h. If the user is currently pressing a button, the corresponding bit position will contain a zero. If the button is up, the corresponding bit will contain a one.

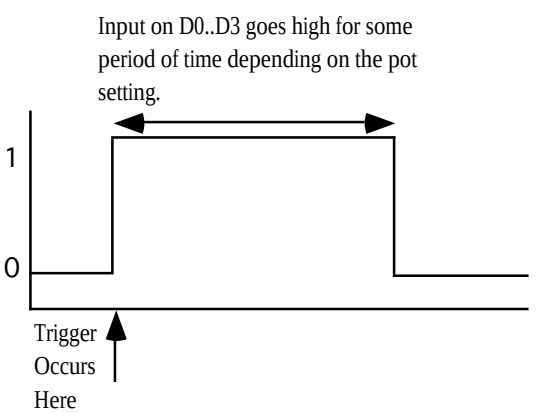
The pot inputs might seem strange at first glance. After all, how can we represent one of a large number of potential pot positions (say, at least 256) with a single bit? Obviously we can't. However, the input bit on this port does not return any type of numeric value specifying the pot position. Instead, each of the

four pot bits is connected to an input of a resistive sensitive 558 quad timer chip. When you trigger the timer chip, it produces an output pulse whose duration is proportional to the resistive input to the timer. The output of this timer chip appears as the input bit for a given pot. The schematic for this circuit is



Joystick Schematic

Normally, the pot input bits contain zero. When you trigger the timer chip, the pot input lines go high for some period of time determined by the current resistance of the potentiometer. By measuring how long this bit stays set, you can get a rough estimate of the resistance. To trigger the pots, simply write any value to I/O port 201h. The actual value you write is unimportant. The following timing diagram shows how the signal varies on each pot's input bit:



Analog Input Timing Signal

The only remaining question is "how do we determine the length of the pulse?" The following short loop demonstrates one way to determine the width of this timing pulse:

```
                mov     cx, -1          ;We're going to count backwards
                mov     dx, 201h        ;Point at joystick port.
                out     dx, al          ;Trigger the timer chip.
CntLp:          in      al, dx          ;Read joystick port.
                test    al, 1           ;Check pot #0 input.
                loopne  CntLp           ;Repeat while high.
                neg     cx              ;Convert CX to a positive value.
```

When this loop finish execution, the cx register will contain the number of passes made through this loop while the timer output signal was a logic one. The larger the value in cx, the longer the pulse and, therefore, the greater the resistance of pot #0.

There are several minor problems with this code. First of all, the code will obviously produce different results on different machines running at different clock rates. For example, a 150 MHz Pentium system will execute this code much faster than a 5 MHz 8088 system[3]. The second problem is that different joysticks and different game adapter cards produce radically different timing results. Even on the same system with the same adapter card and joystick, you may not always get consistent readings on different days. It turns out that the 558 is somewhat temperature sensitive and will produce slightly different readings as the temperature changes.

Unfortunately, there is no way to design a loop like the above so that it returns consistent readings across a wide variety of machines, potentiometers, and game adapter cards. Therefore, you have to write your application software so that it is insensitive to wide variances in the input values from the analog inputs. Fortunately, this is very easy to do, but more on that later.

24.3 Using BIOS' Game I/O Functions

The BIOS provides two functions for reading game adapter inputs. Both are subfunctions of the int 15h handler.

To read the switches, load ah with 84h and dx with zero then execute an int 15h instruction. On return, al will contain the switch readings in the H.O. four bits (see the diagram in the previous section). This function is roughly equivalent to reading port 201h directly.

To read the analog inputs, load ah with 84h and dx with one then execute an int 15h instruction. On return, AX, BX, CX, and DX will contain the values for pots zero, one, two, and three, respectively. In practice, this call should return values in the range 0-400h, though you cannot count on this for reasons described in the previous section.

Very few programs use the BIOS joystick support. It's easier to read the switches directly and reading the pots is not that much more work that calling the BIOS routine. The BIOS code is *very* slow. Most BIOSes read the four pots sequentially, taking up to four times longer than a program that reads all four pots concurrently (see the next section). Because reading the pots can take several hundred microseconds up to several milliseconds, most programmers writing high performance games do not use the BIOS calls, they write their own high performance routines instead.

This is a real shame. By writing drivers specific to the PC's original game adapter design, these developers force the user to purchase and use a standard game adapter card and game input device. Were the game to make the BIOS call, third party developers could create different and unique game controllers and then simply supply a driver that replaces the int 15h routine and provides the same programming interface. For example, Genovation made a device that lets you plug a joystick into the parallel port of a PC.

3. Actually, the speed difference is not as great as you would first think. Joystick adapter cards almost always interface to the computer system via the ISA bus. The ISA bus runs at only 8 Mhz and requires four clock cycles per data transfer (i.e., 500 ns to read the joystick input port). This is equivalent to a small number of wait states on a slow machine and a gigantic number of wait states on a fast machine. Tests run on a 5 MHz 8088 system vs. a 50 MHz 486DX system produces only a 2:1 to 3:1 speed difference between the two machines even though the 486 machine was over 50 times faster for most other computations.

Colorado Spectrum created a similar device that lets you plug a joystick into the serial port. Both devices would let you use a joystick on machines that do not (and, perhaps, cannot) have a game adapter installed. However, games that access the joystick hardware directly will not be compatible with such devices. However, had the game designer made the int 15h call, their software would have been compatible since both Colorado Spectrum and Genovation supply int 15h TSRs to reroute joystick calls to use their devices.

To help overcome game designer's aversion to using the int 15h calls, this text will present a high performance version of the BIOS' joystick code a little later in this chapter. Developers who adopt this *Standard Game Device Interface* will create software that will be compatible with any other device that supports the SGDI standard. For more details, see "The Standard Game Device Interface (SGDI)" on page 1262.

24.4 Writing Your Own Game I/O Routines

Consider again the code that returns some value for a given pot setting:

```
                mov     cx, -1          ;We're going to count backwards
                mov     dx, 201h        ;Point at joystick port.
                out     dx, al          ;Trigger the timer chip.
CntLp:          in      al, dx          ;Read joystick port.
                test    al, 1           ;Check pot #0 input.
                loopne  CntLp           ;Repeat while high.
                neg     cx              ;Convert CX to a positive value.
```

As mentioned earlier, the big problem with this code is that you are going to get wildly different ranges of values from different game adapter cards, input devices, and computer systems. Clearly you cannot count on the code above always producing a value in the range 0..180h under these conditions. Your software will need to dynamically adjust the values it uses depending on the system parameters.

You've probably played a game on the PC where the software asks you to *calibrate* the joystick before use. Calibration generally consists of moving the joystick handle to one corner (e.g., the upper-left corner), pressing a button or key and them moving the handle to the opposite corner (e.g., lower-right) and pressing a button again. Some systems even want you to move the joystick to the center position and press a button as well.

Software that does this is reading the *minimum*, *maximum*, and *centered* values from the joystick. Given at least the minimum and maximum values, you can easily scale any reading to any range you want. By reading the centered value as well, you can get slightly better results, especially on really inexpensive (cheap) joysticks. This process of scaling a reading to a certain range is known as *normalization*. By reading the minimum and maximum values from the user and normalizing every reading thereafter, you can write your programs assuming that the values always fall within a certain range, for example, 0..255. To normalize a reading is very easy, you simply use the following formula:

$$\frac{(CurrentReading - MinimumReading)}{(MaximumReading - MinimumReading)} \times NormalValue$$

The MaximumReading and MinimumReading values are the minimum and maximum values read from the user at the beginning of your application. CurrentReading is the value just read from the game adapter. NormalValue is the upper bounds on the range to which you want to normalize the reading (e.g., 255), the lower bound is always zero[4].

4. If you want a different lower bound, just add whatever value you want fro the lowest value to the result. You will also need to subtract this lower bound from the NormalValue variable in the above equation.

To get better results, especially when using a joystick, you should obtain three readings during the calibration phase for each pot – a minimum value, a maximum value, and a centered value. To normalize a reading when you've got these three values, you would use one of the following formulae:

If the current reading is in the range minimum..center, use this formula:

$$\frac{(Current - Center)}{(Center - Minimum) \times 2} \times NormalValue$$

If the current reading is in the range center..maximum, use this formula:

$$\frac{(Current - Center)}{(Maximum - Center) \times 2} \times NormalValue + \frac{NormalValue}{2}$$

A large number of games on the market today jump through all kinds of hoops trying to coerce joystick readings into a reasonable range. It is surprising how few of them use that simple formula above. Some game designers might argue that the formulae above are overly complex and they are writing high performance games. This is nonsense. It takes two orders of magnitude more time to wait for the joystick to time out than it does to compute the above equations. So use them and make your programs easier to write.

Although normalizing your pot readings takes so little time it is always worthwhile, reading the analog inputs is a very expensive operation in terms of CPU cycles. Since the timer circuit produces relatively fixed time delays for a given resistance, you will waste even more CPU cycles on a fast machine than you do on a slow machine (although reading the pot takes about the same amount of *real* time on any machine). One sure fire way to waste a lot of time is to read several pots one at a time; for example, when reading pots zero and one to get a joystick reading, read pot zero first and then read pot one afterwards. It turns out that you can easily read both pots in parallel. By doing so, you can speed up reading the joystick by a factor of two. Consider the following code:

```
                mov     cx, 1000h       ;Max times through loop
                mov     si, 0           ;We'll put readings in SI and
                mov     di, si          ; di.
                mov     ax, si          ;Set AH to zero.
                mov     dx, 201h        ;Point at joystick port.
                out     dx, al          ;Trigger the timer chip.
CntLp:          in      al, dx          ;Read joystick port.
                and     al, 11b         ;Strip unwanted bits.
                jz      Done
                shr     ax, 1           ;Put pot 0 value into carry.
                adc     si, 0           ;Bump pot 0 value if still active.
                add     di, ax          ;Bump pot 1 value if pot 1 active.
                loop    CntLp           ;Repeat while high.
                and     si, 0FFFh       ;If time-out, force the register(s)
                and     di, 0FFFh       ; containing 1000h to zero.

Done:
```

This code reads both pot zero and pot one at the same time. It works by looping while either pot is active[5]. Each time through the loop, this code adds the pots' bit values to separate register that accumulator the result. When this loop terminates, `si` and `di` contain the readings for both pots zero and one.

Although this particular loop contains more instructions than the previous loop, it still takes the same amount of time to execute. Remember, the output pulses on the 558 timer determine how long this code takes to execute, the number of instructions in the loop contribute very little to the execution time. However, the time this loop takes to execute one iteration of the loop does effect the *resolution* of this joystick read routine. The faster the loop executes, the more iterations the loop will run during the same timing period and the finer will be the measurement. Generally, though, the resolution of the above code is much greater than the accuracy of the electronics and game input device, so this isn't much of a concern.

5. This code provides a time-out feature in the event there is no game adapter installed. In such an event this code forces the readings to zero.

The code above demonstrates how to read two pots. It is very easy to extend this code to read three or four pots. An example of such a routine appears in the section on the SGDI device driver for the standard game adapter card.

The other game device input, the switches, would seem to be simple in comparison to the potentiometer inputs. As usual, things are not as easy as they would seem at first glance. The switch inputs have some problems of their own.

The first issue is keybounce. The switches on a typical joystick are probably an order of magnitude worse than the keys on the cheapest keyboard. Keybounce, and lots of it, is a fact you're going to have to deal with when reading joystick switches. In general, you shouldn't read the joystick switches more often than once every 10 msec. Many games read the switches on the 55 msec timer interrupt. For example, suppose your timer interrupt reads the switches and stores the result in a memory variable. The main application, when wanting to fire a weapon, checks the variable. If it's set, the main program clears the variable and fires the weapon. Fifty-five milliseconds later, the timer sets the button variable again and the main program will fire again the next time it checks the variable. Such a scheme will totally eliminate the problems with keybounce.

The technique above solves another problem with the switches: keeping track of when the button first goes down. Remember, when you read the switches, the bits that come back tell you that the switch is currently down. It does not tell you that the button was just pressed. You have to keep track of this yourself. One easy way to detect when a user first presses a button is to save the previous switch reading and compare it against the current reading. If they are different and the current reading indicates a switch depression, then this is a new switch down.

24.5 The Standard Game Device Interface (SGDI)

The Standard Game Device Interface (SGDI) is a specification for an int 15h service that lets you read an arbitrary number of pots and joysticks. Writing SGDI compliant applications is easy and helps make your software compatible with any game device which provides SGDI compliance. By writing your applications to use the SGDI API you can ensure that your applications will work with future devices that provide extended SGDI capability. To understand the power and extensibility of the SGDI, you need to take a look at the *application programmer's interface* (API) for the SGDI.

24.5.1 Application Programmer's Interface (API)

The SGDI interface extends the PC's joystick BIOS int 15h API. You make SGDI calls by loading the 80x86 ah register with 84h and dx with an appropriate SGDI function code and then executing an int 15h instruction. The SGDI interface simply extends the functionality of the built-in BIOS routines. Note that and program that calls the standard BIOS joystick routines will work with an SGDI driver. The following table lists each of the SGDI functions:

Table 87: SGDI Functions and API (int 15h, ah=84h)

DH	Inputs	Outputs	Description
00	dl = 0	al- Switch readings	Read4Sw. This is the standard BIOS subfunction zero call. This reads the status of the first four switches and returns their values in the upper four bits of the al register.
00	dl = 1	ax- pot 0 bx- pot 1 cx- pot 2 dx- pot 3	Read4Pots. Standard BIOS subfunction one call. Reads all four pots (concurrently) and returns their raw values in ax, bx, cx, and dx as per BIOS specifications.

Table 87: SGDI Functions and API (int 15h, ah=84h)

DH	Inputs	Outputs	Description
01	dl = pot #	al= pot reading	ReadPot. This function reads a pot and returns a *normalized* reading in the range 0..255.
02	dl = 0 al = pot mask	al = pot 0 ah = pot 1 dl = pot 2 dh = pot 3	Read4. This routine reads the four pots on the standard game adapter card just like the Read4Pots function above. However, this routine normalizes the four values to the range 0..255 and returns those values in al, ah, dl, and dh. On entry, the al register contains a "pot mask" that you can use to select which of the four pots this routine actually reads.
03	dl = pot # al = minimum bx= maximum cx= centered		Calibrate. This function calibrates the pots for those calls that return normalized values. You must calibrate the pots before calling any such pot functions (ReadPot and Read4 above). The input values must be *raw* pot readings obtained by Read4Pots or other function that returns raw values.
04	dl = pot #	al = 0 if not calibrated, 1 if calibrated.	TestPotCalibrate. Checks to see if the specified pot has already been calibrated. Returns an appropriate value in al denoting the calibration status for the specified pot. See the note above about the need for calibration.
05	dl = pot #	ax = raw value	ReadRaw. Reads a raw value from the specified pot. You can use this call to get the raw values required by the calibrate routine, above.
08	dl= switch #	ax = switch value	ReadSw. Read the specified switch and returns zero (switch up) or one (switch down) in the ax register.
09		ax = switch values	Read16Sw. This call lets an application read up to 16 switches on a game device at a time. Bit zero of ax corresponds to switch zero, bit 15 of ax corresponds to switch fifteen.
80h			Remove. This function removes the driver from memory. Application programs generally won't make this call.
81h			TestPresence. This routine returns zero in the ax register if an SGDI driver is present in memory. It returns ax's value unchanged otherwise (in particular, ah will still contain 84h).

24.5.2 Read4Sw

Inputs: ah= 84h, dx = 0

This is the standard BIOS read switches call. It returns the status switches zero through three on the joystick in the upper four bits of the al register. Bit four corresponds to switch zero, bit five to switch one, bit six to switch two, and bit seven to switch three. One zero in each bit position denotes a depressed switch, a one bit corresponds to a switch in the up position. This call is provided for compatibility with the existing BIOS joystick routines. To read the joystick switches you should use the Read16Sw call described later in this document.

24.5.3 Read4Pots:

Inputs: ah= 84h, dx = 1

This is the standard BIOS read pots call. It reads the four pots on the standard game adapter card and returns their readings in the ax (x axis/pot 0), bx (y axis/pot 1), cx (pot 2), and dx (pot 3) registers. These are *raw, uncalibrated*, pot readings whose values will differ from machine to machine and vary depending upon the game I/O card in use. This call is provided for compatibility with the existing BIOS

joystick routines. To read the pots you should use the ReadPot, Read4, or ReadRaw routines described in the next several sections.

24.5.4 ReadPot

Inputs: ah=84h, dh=1, dl=Pot number.

This reads the specified pot and returns a *normalized* pot value in the range 0..255 in the al register. This routine also sets ah to zero. Although the SGDI standard provides for up to 255 different pots, most adapters only support pots zero, one, two, and three. If you attempt to read any nonsupported pot this function returns zero in ax. Since the values are normalized, this call returns comparable values for a given game control setting regardless of machine, clock frequency, or game I/O card in use. For example, a reading of 128 corresponds (roughly) to the center setting on almost any machine. To properly produce normalized results, you must *calibrate* a given pot before making this call. See the CalibratePot routine for more details.

24.5.5 Read4:

Inputs: ah = 84h, al = pot mask, dx=0200h

This routine reads the four pots on the game adapter card, just like the BIOS call (Read4Pots). However, it returns normalized values in al (x axis/pot 0), ah (y axis/pot 1), dl (pot 2), and dh (pot 3). Since this routine returns normalized values between zero and 255, you must calibrate the pots before calling this code. The al register contains a "pot mask" value. The L.O. four bits of al determine if this routine will actually read each pot. If bit zero, one, two, or three is one, then this function will read the corresponding pot; if the bits are zero, this routine will not read the corresponding pot and will return zero in the corresponding register.

24.5.6 CalibratePot

Inputs: ah=84h, dh=3, dl=pot #, al=minimum value, bx=maximum value, cx=centered value.

Before you attempt to read a pot with the ReadPot or Read4 routines, you need to calibrate that pot. If you read a pot without first calibrating it, the SGDI driver will return only zero for that pot reading. To calibrate a pot you will need to read raw values for the pot in a minimum position, maximum position, and a centered position[6]. *These must be raw pot readings.* Use readings obtained by the Read4Pots routine. In theory, you need only calibrate a pot once after loading the SGDI driver. However, temperature fluctuations and analog circuitry drift may decalibrate a pot after considerable use. Therefore, you should recalibrate the pots you intend to read each time the user runs your application. Furthermore, you should give the user the option of recalibrating the pots at any time within your program.

24.5.7 TestPotCalibration

Inputs: ah= 84h, dh=4 , dl = pot #.

This routine returns zero or one in ax denoting *not calibrated* or *calibrated*, respectively. You can use the call to see if the pots you intend to use have already been calibrated and you can skip the calibration phase. Please, however, note the comments about drift in the previous paragraph.

6. Many programmers compute the centered value as the arithmetic mean of the minimum and maximum values.

24.5.8 ReadRaw

Inputs: ah = 84h, dh = 5, dl = pot #

Reads the specified pot and returns a raw (not calibrated) value in ax. You can use this routine to obtain minimum, centered, and maximum values for use when calling the calibrate routine.

24.5.9 ReadSwitch

Inputs: ah= 84h, dh = 8, dl = switch #

This routine reads the specified switch and returns zero in ax if the switch is *not* depressed. It returns one if the switch is depressed. Note that this value is opposite the bit settings the Read4Sw function returns.

If you attempt to read a switch number for an input that is not available on the current device, the SGDI driver will return zero (switch up). Standard game devices only support switches zero through three and most joysticks only provide two switches. Therefore, unless you are willing to tie your application to a specific device, you shouldn't use any switches other than zero or one.

24.5.10 Read16Sw

Inputs: ah = 84h, dh = 9

This SGDI routine reads up to sixteen switches with a single call. It returns a bit vector in the ax register with bit 0 corresponding to switch zero, bit one corresponding to switch one, etc. Ones denote switch depressed and zeros denote switches not depressed. Since the standard game adapter only supports four switches, only bits zero through three of a1 contain meaningful data (for those devices). All other bits will always contain zero. SGDI drivers for the CH Product's Flightstick Pro and Thrustmaster joysticks will return bits for the entire set of switches available on those devices.

24.5.11 Remove

Inputs: ah= 84h, dh= 80h

This call will attempt to remove the SGDI driver from memory. Generally, only the SGDI.EXE code itself would invoke this routine. You should use the TestPresence routine (described next) to see if the driver was actually removed from memory by this call.

24.5.12 TestPresence

Inputs: ah=84h, dh=81h

If an SGDI driver is present in memory, this routine return ax=0 and a pointer to an identification string in es:bx. If an SGDI driver is not present, this call will return ax unchanged.

24.5.13 An SGDI Driver for the Standard Game Adapter Card

If you write your program to make SGDI calls, you will discover that the TestPresence call will probably return "not present" when your program searches for a resident SGDI driver in memory. This is because few manufacturers provide SGDI drivers at this point and even fewer standard game adapter

companies ship any software at all with their products, much less an SGDI driver. Gee, what kind of standard is this if no one uses it? Well, the purpose of this section is to rectify that problem.

The assembly code that appears at the end of this section provides a fully functional, public domain, SGDI driver for the standard game adapter card (the next section present an SGDI driver for the CH Products' Flightstick Pro). This allows you to write your application making only SGDI calls. By supplying the SGDI TSR with your product, your customers can use your software with all standard joysticks. Later, if they purchase a specialized device with its own SGDI driver, your software will automatically work with that driver with no changes to your software[7].

If you do not like the idea of having a user run a TSR before your application, you can always include the following code within your program's code space and activate it if the SGDI TestPresence call determines that no other SGDI driver is present in memory when you start your program.

Here's the complete code for the standard game adapter SGDI driver:

```
                .286
                page    58, 132
                name    SGDI
                title   SGDI Driver for Standard Game Adapter Card
                subttl  This Program is Public Domain Material.

; SGDI.EXE
;
;       Usage:
;               SDGI
;
; This program loads a TSR which patches INT 15 so arbitrary game programs
; can read the joystick in a portable fashion.
;
;
; We need to load cseg in memory before any other segments!

cseg            segment     para public 'code'
cseg            ends


; Initialization code, which we do not need except upon initial load,
; goes in the following segment:

Initialize      segment     para public 'INIT'
Initialize      ends

; UCR Standard Library routines which get dumped later on.

                .xlist
                include         stdlib.a
                includelib stdlib.lib
                .list

sseg            segment     para stack 'stack'
sseg            ends

zzzzzzseg       segment     para public 'zzzzzzseg'
zzzzzzseg       ends



CSEG            segment     para public 'CODE'
                assume      cs:cseg, ds:nothing

wp              equ         <word ptr>
byp             equ         <byte ptr>

Int15Vect       dword       0

PSP             word        ?
```

7. Of course, your software may not take advantage of extra features, like additional switches and pots, but at least your software will support the standard set of features on that device.

```
; Port addresses for a typical joystick card:

JoyPort       equ       201h
JoyTrigger    equ       201h

; Data structure to hold information about each pot.
; (mainly for calibration and normalization purposes).

Pot           struc
PotMask       byte      0               ;Pot mask for hardware.
DidCal        byte      0               ;Is this pot calibrated?
min           word      5000            ;Minimum pot value
max           word      0               ;Max pot value
center        word      0               ;Pot value in the middle
Pot           ends

; Variables for each of the pots. Must initialize the masks so they
; mask out all the bits except the incomming bit for each pot.

Pot0          Pot       <1>
Pot1          Pot       <2>
Pot2          Pot       <4>
Pot3          Pot       <8>



; The IDstring address gets passed back to the caller on a testpresence
; call. The four bytes before the IDstring must contain the serial number
; and current driver number.

SerialNumber  byte      0,0,0
IDNumber      byte      0
IDString      byte      “Standard SGDI Driver”,0
              byte      “Public Domain Driver Written by Randall L. Hyde”,0


;=============================================================================
;
; ReadPots-   AH contains a bit mask to determine which pots we should read.
;              Bit 0 is one if we should read pot 0, bit 1 is one if we should
;              read pot 1, bit 2 is one if we should read pot 2, bit 3 is one
;              if we should read pot 3. All other bits will be zero.
;
;       This code returns the pot values in SI, BX, BP, and DI for Pot 0, 1,
;       2, & 3.
;

ReadPots      proc      near
              sub       bp, bp
              mov       si, bp
              mov       di, bp
              mov       bx, bp

; Wait for any previous signals to finish up before trying to read this
; guy. It is possible that the last pot we read was very short. However,
; the trigger signal starts timers running for all four pots. This code
; terminates as soon as the current pot times out. If the user immediately
; reads another pot, it is quite possible that the new pot’s timer has
; not yet expired from the previous read. The following loop makes sure we
; aren’t measuring the time from the previous read.

              mov       dx, JoyPort
              mov       cx, 400h
Wait4Clean:   in        al, dx
              and       al, 0Fh
              loopnz    Wait4Clean

; Okay, read the pots. The following code triggers the 558 timer chip
; and then sits in a loop until all four pot bits (masked with the pot mask
; in AL) become zero. Each time through this loop that one or more of these
; bits contain zero, this loop increments the corresponding register(s).

              mov       dx, JoyTrigger
```

```
                out     dx, al          ;Trigger pots
                mov     dx, JoyPort
                mov     cx, 1000h       ;Don't let this go on forever.
PotReadLoop:    in      al, dx
                and     al, ah
                jz      PotReadDone
                shr     al, 1
                adc     si, 0           ;Increment SI if pot 0 still active.
                shr     al, 1
                adc     bx, 0       ;Increment BX if pot 1 still active.
                shr     al, 1
                adc     bp, 0       ;Increment BP if pot 2 still active.
                shr     al, 1
                adc     di, 0       ;Increment DI if pot 3 still active.
                loop    PotReadLoop ;Stop, eventually, if funny hardware.

                and     si, 0FFFh   ;If we drop through to this point,
                and     bx, 0FFFh   ; one or more pots timed out (usually
                and     bp, 0FFFh   ; because they are not connected).
                and     di, 0FFFh   ; The reg contains 4000h, set it to 0.
PotReadDone:    ret
ReadPots        endp



;-----------------------------------------------------------------------------
;
; Normalize-    BX contains a pointer to a pot structure, AX contains
;               a pot value. Normalize that value according to the
;               calibrated pot.
;
; Note: DS must point at cseg before calling this routine.


                assume  ds:cseg
Normalize       proc    near
                push    cx

; Sanity check to make sure the calibration process went okay.

                cmp     [bx].Pot.DidCal, 0 ;Is this pot calibrated?
                je      BadNorm         ;If not, quit.

                mov     dx, [bx].Pot.Center ;Do a sanity check on the
                cmp     dx, [bx].Pot.Min  ; min, center, and max
                jbe     BadNorm         ; values to make sure
                cmp     dx, [bx].Pot.Max  ; min < center < max.
                jae     BadNorm

; Clip the value if it is out of range.

                cmp     ax, [bx].Pot.Min  ;If the value is less than
                ja      MinOkay         ; the minimum value, set it
                mov     ax, [bx].Pot.Min  ; to the minimum value.
MinOkay:

                cmp     ax, [bx].Pot.Max  ;If the value is greater than
                jb      MaxOkay         ; the maximum value, set it
                mov     ax, [bx].Pot.Max  ; to the maximum value.
MaxOkay:

; Scale this guy around the center:

                cmp     ax, [bx].Pot.Center ;See if less than or greater
                jb      Lower128        ; than centered value.

; Okay, current reading is greater than the centered value, scale the reading
; into the range 128..255 here:

                sub     ax, [bx].Pot.Center
                mov     dl, ah          ;Multiply by 128
                mov     ah, al
                mov     dh, 0
                mov     al, dh
```

```
                shr     dl, 1
                rcr     ax, 1
                mov     cx, [bx].Pot.Max
                sub     cx, [bx].Pot.Center
                jz      BadNorm         ;Prevent division by zero.
                div     cx              ;Compute normalized value.
                add     ax, 128         ;Scale to range 128..255.
                cmp     ah, 0
                je      NormDone
                mov     ax, 0ffh        ;Result must fit in 8 bits!
                jmp     NormDone

; If the reading is below the centered value, scale it into the range
; 0..127 here:

Lower128:       sub     ax, [bx].Pot.Min
                mov     dl, ah
                mov     ah, al
                mov     dh, 0
                mov     al, dh
                shr     dl, 1
                rcr     ax, 1
                mov     cx, [bx].Pot.Center
                sub     cx, [bx].Pot.Min
                jz      BadNorm
                div     cx
                cmp     ah, 0
                je      NormDone
                mov     ax, 0ffh
                jmp     NormDone

; If something went wrong, return zero as the normalized value.

BadNorm:        sub     ax, ax

NormDone:       pop     cx
                ret
Normalize       endp
                assume  ds:nothing

;============================================================================
; INT 15h handler functions.
;============================================================================
;
; Although these are defined as near procs, they are not really procedures.
; The MyInt15 code jumps to each of these with BX, a far return address, and
; the flags sitting on the stack. Each of these routines must handle the
; stack appropriately.
;
;----------------------------------------------------------------------------
; BIOS-  Handles the two BIOS calls, DL=0 to read the switches, DL=1 to
;        read the pots. For the BIOS routines, we'll ignore the cooley
;        switch (the hat) and simply read the other four switches.

BIOS            proc    near
                cmp     dl, 1           ;See if switch or pot routine.
                jb      Read4Sw
                je      ReadBIOSPots

; If not a valid BIOS call, jump to the original INT 15 handler and
; let it take care of this call.

                pop     bx
                jmp     cs:Int15Vect    ;Let someone else handle it!

; BIOS read switches function.

Read4Sw:        push    dx
                mov     dx, JoyPort
                in      al, dx
                and     al, 0F0h        ;Return only switch values.
                pop     dx
                pop     bx
                iret
```

```
; BIOS read pots function.

ReadBIOSPots:  pop       bx                ;Return a value in BX!
               push      si
               push      di
               push      bp
               mov       ah, 0Fh           ;Read all four pots.
               call      ReadPots
               mov       ax, si
               mov       cx, bp            ;BX already contains pot 1 reading.
               mov       dx, di
               pop       bp
               pop       di
               pop       si
               iret
BIOS           endp

;-----------------------------------------------------------------------------
;
; ReadPot-     On entry, DL contains a pot number to read.
;              Read and normalize that pot and return the result in AL.

               assume    ds:cseg
ReadPot        proc      near
;;;;;;;;;;     push      bx                ;Already on stack.
               push      ds
               push      cx
               push      dx
               push      si
               push      di
               push      bp

               mov       bx, cseg
               mov       ds, bx

; If dl = 0, read and normalize the value for pot 0, if not, try some
; other pot.

               cmp       dl, 0
               jne       Try1
               mov       ah, Pot0.PotMask  ;Get bit for this pot.
               call      ReadPots          ;Read pot 0.
               lea       bx, Pot0          ;Pointer to pot data.
               mov       ax, si            ;Get pot 0 reading.
               call      Normalize         ;Normalize to 0..FFh.
               jmp       GotPot            ;Return to caller.

; Test for DL=1 here (read and normalize pot 1).

Try1:          cmp       dl, 1
               jne       Try2
               mov       ah, Pot1.PotMask
               call      ReadPots
               mov       ax, bx
               lea       bx, Pot1
               call      Normalize
               jmp       GotPot

; Test for DL=2 here (read and normalize pot 2).

Try2:          cmp       dl, 2
               jne       Try3
               mov       ah, Pot2.PotMask
               call      ReadPots
               lea       bx, Pot2
               mov       ax, bp
               call      Normalize
               jmp       GotPot

; Test for DL=3 here (read and normalize pot 3).

Try3:          cmp       dl, 3
               jne       BadPot
```

```
                mov     ah, Pot3.PotMask
                call    ReadPots
                lea     bx, Pot3
                mov     ax, di
                call    Normalize
                jmp     GotPot

; Bad value in DL if we drop to this point. The standard game card
; only supports four pots.

BadPot:         sub     ax, ax          ;Pot not available, return zero.
GotPot:         pop     bp
                pop     di
                pop     si
                pop     dx
                pop     cx
                pop     ds
                pop     bx
                iret
ReadPot         endp
                assume  ds:nothing


;-----------------------------------------------------------------------------
;
; ReadRaw-      On entry, DL contains a pot number to read.
;               Read that pot and return the unnormalized result in AX.

                assume  ds:cseg
ReadRaw         proc    near
;;;;;;;;;;      push    bx              ;Already on stack.
                push    ds
                push    cx
                push    dx
                push    si
                push    di
                push    bp

                mov     bx, cseg
                mov     ds, bx

; This code is almost identical to the ReadPot code. The only difference
; is that we don't bother normalizing the result and (of course) we return
; the value in AX rather than AL.

                cmp     dl, 0
                jne     Try1
                mov     ah, Pot0.PotMask
                call    ReadPots
                mov     ax, si
                jmp     GotPot

Try1:           cmp     dl, 1
                jne     Try2
                mov     ah, Pot1.PotMask
                call    ReadPots
                mov     ax, bx
                jmp     GotPot

Try2:           cmp     dl, 2
                jne     Try3
                mov     ah, Pot2.PotMask
                call    ReadPots
                mov     ax, bp
                jmp     GotPot

Try3:           cmp     dl, 3
                jne     BadPot
                mov     ah, Pot3.PotMask
                call    ReadPots
                mov     ax, di
                jmp     GotPot

BadPot:         sub     ax, ax          ;Pot not available, return zero.
```

```
GotPot:         pop         bp
                pop         di
                pop         si
                pop         dx
                pop         cx
                pop         ds
                pop         bx
                iret
ReadRaw         endp
                assume      ds:nothing


;-----------------------------------------------------------------------------
; Read4Pots-    Reads pots zero, one, two, and three returning their
;               values in AL, AH, DL, and DH.
;
;               On entry, AL contains the pot mask to select which pots
;               we should read (bit 0=1 for pot 0, bit 1=1 for pot 1, etc).

Read4Pots       proc        near
;;;;;;;;;;;     push        bx                  ;Already on stack
                push        ds
                push        cx
                push        si
                push        di
                push        bp

                mov         dx, cseg
                mov         ds, dx

                mov         ah, al
                call        ReadPots

                push        bx                  ;Save pot 1 reading.
                mov         ax, si              ;Get pot 0 reading.
                lea         bx, Pot0            ;Point bx at pot0 vars.
                call        Normalize           ;Normalize.
                mov         cl, al              ;Save for later.

                pop         ax                  ;Retreive pot 1 reading.
                lea         bx, Pot1
                call        Normalize
                mov         ch, al              ;Save normalized value.

                mov         ax, bp
                lea         bx, Pot2
                call        Normalize
                mov         dl, al              ;Pot 2 value.

                mov         ax, di
                lea         bx, Pot3
                call        Normalize
                mov         dh, al              ;Pot 3 value.
                mov         ax, cx              ;Pots 0 and 1.

                pop         bp
                pop         di
                pop         si
                pop         cx
                pop         ds
                pop         bx
                iret
Read4Pots       endp



;-----------------------------------------------------------------------------
; CalPot-       Calibrate the pot specified by DL. On entry, AL contains
;               the minimum pot value (it better be less than 256!), BX
;               contains the maximum pot value, and CX contains the centered
;               pot value.

                assume      ds:cseg
```

```
CalPot          proc    near
                pop     bx              ;Retrieve maximum value
                push    ds
                push    si
                mov     si, cseg
                mov     ds, si

; Sanity check on parameters, sort them in ascending order:

                mov     ah, 0
                cmp     bx, cx          ;Make sure center < max
                ja      GoodMax
                xchg    bx, cx
GoodMax:        cmp     ax, cx          ;Make sure min < center.
                jb      GoodMin         ; (note: may make center<max).
                xchg    ax, cx
GoodMin:        cmp     cx, bx          ;Again, be sure center < max.
                jb      GoodCenter
                xchg    cx, bx
GoodCenter:


; Okay, figure out who were supposed to calibrate:

                lea     si, Pot0
                cmp     dl, 1
                jb      DoCal           ;Branch if this is pot 0
                lea     si, Pot1
                je      DoCal           ;Branch if this is pot 1
                lea     si, Pot2
                cmp     dl, 3
                jb      DoCal           ;Branch if this is pot 2
                jne     CalDone         ;Branch if not pot 3
                lea     si, Pot3

DoCal:          mov     [si].Pot.min, ax  ;Store away the minimum,
                mov     [si].Pot.max, bx  ; maximum, and
                mov     [si].Pot.center, cx ; centered values.
                mov     [si].Pot.DidCal, 1  ;Note we've cal'd this pot.
CalDone:        pop     si
                pop     ds
                iret
CalPot          endp
                assume  ds:nothing


;-----------------------------------------------------------------------------
; TestCal-      Just checks to see if the pot specified by DL has already
;               been calibrated.

                assume  ds:cseg
TestCal         proc    near
;;;;;;;;        push    bx              ;Already on stack
                push    ds
                mov     bx, cseg
                mov     ds, bx

                sub     ax, ax          ;Assume no calibration (also zeros AH)
                lea     bx, Pot0        ;Get the address of the specified
                cmp     dl, 1           ; pot's data structure into the
                jb      GetCal          ; BX register.
                lea     bx, Pot1
                je      GetCal
                lea     bx, Pot2
                cmp     dl, 3
                jb      GetCal
                jne     BadCal
                lea     bx, Pot3

GetCal:         mov     al, [bx].Pot.DidCal
BadCal:         pop     ds
                pop     bx
                iret
TestCal         endp
```

```
                assume  ds:nothing


;-----------------------------------------------------------------------------
;
; ReadSw-       Reads the switch whose switch number appears in DL.

ReadSw          proc    near
;;;;;;;         push    bx                  ;Already on stack
                push    cx

                sub     ax, ax          ;Assume no such switch.
                cmp     dl, 3           ;Return if the switch number is
                ja      NotDown         ; greater than three.

                mov     cl, dl          ;Save switch to read.
                add     cl, 4           ;Move from position four down to zero.
                mov     dx, JoyPort
                in      al, dx          ;Read the switches.
                shr     al, cl          ;Move desired switch bit into bit 0.
                xor     al, 1           ;Invert so sw down=1.
                and     ax, 1           ;Remove other junk bits.
NotDown:        pop     cx
                pop     bx
                iret
ReadSw          endp


;-----------------------------------------------------------------------------
;
; Read16Sw-     Reads all four switches and returns their values in AX.

Read16Sw        proc    near
;;;;;;;;        push    bx                  ;Already on stack
                mov     dx, JoyPort
                in      al, dx
                shr     al, 4
                xor     al, 0Fh             ;Invert all switches.
                and     ax, 0Fh             ;Set other bits to zero.
                pop     bx
                iret
Read16Sw        endp


;*****************************************************************************
;
; MyInt15-      Patch for the BIOS INT 15 routine to control reading the
;               joystick.

MyInt15         proc    far
                push    bx
                cmp     ah, 84h             ;Joystick code?
                je      DoJoystick
OtherInt15:     pop     bx
                jmp     cs:Int15Vect

DoJoystick:     mov     bh, 0
                mov     bl, dh
                cmp     bl, 80h
                jae     VendorCalls
                cmp     bx, JmpSize
                jae     OtherInt15
                shl     bx, 1
                jmp     wp cs:jmptable[bx]

jmptable        word    BIOS
                word    ReadPot, Read4Pots, CalPot, TestCal
                word    ReadRaw, OtherInt15, OtherInt15
                word    ReadSw, Read16Sw
JmpSize         =       ($-jmptable)/2


; Handle vendor specific calls here.
```

```
VendorCalls:  je        RemoveDriver
              cmp       bl, 81h
              je        TestPresence
              pop       bx
              jmp       cs:Int15Vect


; TestPresence- Returns zero in AX and a pointer to the ID string in ES:BX

TestPresence: pop       bx              ;Get old value off stack.
              sub       ax, ax
              mov       bx, cseg
              mov       es, bx
              lea       bx, IDString
              iret

; RemoveDriver-If there are no other drivers loaded after this one in
;               memory, disconnect it and remove it from memory.

RemoveDriver:
              push      ds
              push      es
              push      ax
              push      dx

              mov       dx, cseg
              mov       ds, dx

; See if we're the last routine patched into INT 15h

              mov       ax, 3515h
              int       21h
              cmp       bx, offset MyInt15
              jne       CantRemove
              mov       bx, es
              cmp       bx, wp seg MyInt15
              jne       CantRemove

              mov       ax, PSP         ;Free the memory we're in
              mov       es, ax
              push      es
              mov       ax, es:[2ch]    ;First, free env block.
              mov       es, ax
              mov       ah, 49h
              int       21h

              pop       es              ;Now free program space.
              mov       ah, 49h
              int       21h

              lds       dx, Int15Vect   ;Restore previous int vect.
              mov       ax, 2515h
              int       21h

CantRemove:   pop       dx
              pop       ax
              pop       es
              pop       ds
              pop       bx
              iret
MyInt15       endp
cseg          ends


Initialize    segment   para public 'INIT'
              assume    cs:Initialize, ds:cseg
Main          proc
              mov       ax, cseg        ;Get ptr to vars segment
              mov       es, ax
              mov       es:PSP, ds      ;Save PSP value away
              mov       ds, ax

              mov       ax, zzzzzzseg
```

```
                mov     es, ax
                mov     cx, 100h
                meminit2

                print
                byte    " Standard Game Device Interface driver",cr,lf
                byte    " PC Compatible Game Adapter Cards",cr,lf
                byte    " Written by Randall Hyde",cr,lf
                byte    cr,lf
                byte    cr,lf
                byte    "'SGDI REMOVE' removes the driver from memory",cr,lf
                byte    lf
                byte    0

                mov     ax, 1
                argv                    ;If no parameters, empty str.
                stricmpl
                byte    "REMOVE",0
                jne     NoRmv

                mov     dh, 81h         ;Remove opcode.
                mov     ax, 84ffh
                int     15h             ;See if we're already loaded.
                test    ax, ax          ;Get a zero back?
                jz      Installed
                print
                byte    "SGDI driver is not present in memory, REMOVE "
                byte    "command ignored.",cr,lf,0
                mov     ax, 4c01h;Exit to DOS.
                int     21h

Installed:      mov     ax, 8400h
                mov     dh, 80h         ;Remove call
                int     15h
                mov     ax, 8400h
                mov     dh, 81h         ;TestPresence call
                int     15h
                cmp     ax, 0
                je      NotRemoved
                print
                byte    "Successfully removed SGDI driver from memory."
                byte    cr,lf,0
                mov     ax, 4c01h       ;Exit to DOS.
                int     21h

NotRemoved:     print
                byte    "SGDI driver is still present in memory.",cr,lf,0
                mov     ax, 4c01h       ;Exit to DOS.
                int     21h




; Okay, Patch INT 15 and go TSR at this point.

NoRmv:
                mov     ax, 3515h
                int     21h
                mov     wp Int15Vect, bx
                mov     wp Int15Vect+2, es

                mov     dx, cseg
                mov     ds, dx
                mov     dx, offset MyInt15
                mov     ax, 2515h
                int     21h

                mov     dx, cseg
                mov     ds, dx
                mov     dx, seg Initialize
                sub     dx, ds:psp
                add     dx, 2
                mov     ax, 3100h       ;Do TSR
```

```
                int         21h
Main            endp

Initialize      ends


sseg            segment     para stack 'stack'
                word        128 dup (0)
endstk          word        ?
sseg            ends


zzzzzzseg       segment     para public 'zzzzzzseg'
                byte        16 dup (0)
zzzzzzseg       ends
                end         Main
```

The following program makes several different types of calls to an SGDI driver. You can use this code to test out an SGDI TSR:

```
                .xlist
                include     stdlib.a
                includelib stdlib.lib
                .list


cseg            segment     para public 'code'
                assume      cs:cseg, ds:nothing

MinVal0         word        ?
MinVal1         word        ?
MaxVal0         word        ?
MaxVal1         word        ?


; Wait4Button- Waits until the user presses and releases a button.

Wait4Button     proc        near
                push        ax
                push        dx
                push        cx

W4BLp:          mov         ah, 84h
                mov         dx, 900h        ;Read the L.O. 16 buttons.
                int         15h
                cmp         ax, 0           ;Any button down? If not,
                je          W4BLp           ; loop until this is so.

                xor         cx, cx          ;Debouncing delay loop.
Delay:          loop        Delay

W4nBLp:         mov         ah, 84h         ;Now wait until the user releases
                mov         dx, 900h        ; all buttons
                int         15h
                cmp         ax, 0
                jne         W4nBLp

Delay2:         loop        Delay2

                pop         cx
                pop         dx
                pop         ax
                ret
Wait4Button     endp

Main            proc

                print
                byte        "SGDI Test Program.",cr,lf
```

```
                byte    "Written by Randall Hyde",cr,lf,lf
                byte    "Press any key to continue",cr,lf,0

                getc

                mov     ah, 84h
                mov     dh, 4           ;Test presence call.
                int     15h
                cmp     ax, 0           ;See if there
                je      MainLoop0
                print
                byte    "No SGDI driver present in memory.",cr,lf,0
                jmp     Quit

MainLoop0:print
                byte    "BIOS: ",0

; Okay, read the switches and raw pot values using the BIOS compatible calls.

                mov     ah, 84h
                mov     dx, 0           ;BIOS compat. read switches.
                int     15h
                puth                    ;Output switch values.
                mov     al, ' '
                putc

                mov     ah, 84h         ;BIOS compat. read pots.
                mov     dx, 1
                int     15h
                putw
                mov     al, ' '
                putc
                mov     ax, bx
                putw
                mov     al, ' '
                putc
                mov     ax, cx
                putw
                mov     al, ' '
                putc
                mov     ax, dx
                putw

                putcr
                mov     ah, 1           ;Repeat until key press.
                int     16h
                je      MainLoop0
                getc


; Read the minimum and maximum values for each pot from the user so we
; can calibrate the pots.

                print
                byte    cr,lf,lf,lf
                byte    "Move joystick to upper left corner and press "
                byte    "any button.",cr,lf,0

                call    Wait4Button
                mov     ah, 84h
                mov     dx, 1           ;Read Raw Values
                int     15h
                mov     MinVal0, ax
                mov     MinVal1, bx

                print
                byte    cr,lf
                byte    "Move the joystick to the lower right corner "
                byte    "and press any button",cr,lf,0

                call    Wait4Button
                mov     ah, 84h
                mov     dx, 1           ;Read Raw Values
                int     15h
```

```
                mov     MaxVal0, ax
                mov     MaxVal1, bx

; Calibrate the pots.

                mov     ax, MinVal0;Will be eight bits or less.
                mov     bx, MaxVal0
                mov     cx, bx          ;Compute centered value as the
                add     cx, ax          ; average of these two (this is
                shr     cx, 1           ; dangerous, but usually works!)
                mov     ah, 84h
                mov     dx, 300h;Calibrate pot 0
                int     15h

                mov     ax, MinVal1;Will be eight bits or less.
                mov     bx, MaxVal1
                mov     cx, bx          ;Compute centered value as the
                add     cx, ax          ; average of these two (this is
                shr     cx, 1           ; dangerous, but usually works!)
                mov     ah, 84h
                mov     dx, 301h        ;Calibrate pot 1
                int     15h

MainLoop1:      print
                byte    "ReadSw: ",0

; Okay, read the switches and raw pot values using the BIOS compatible calls.

                mov     ah, 84h
                mov     dx, 800h        ;Read switch zero.
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 801h        ;Read switch one.
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 802h        ;Read switch two.
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 803h        ;Read switch three.
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 804h        ;Read switch four
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 805h        ;Read switch five.
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 806h        ;Read switch six.
                int     15h
                or      al, '0'
                putc

                mov     ah, 84h
                mov     dx, 807h        ;Read switch seven.
                int     15h             ;We won't bother with
                or      al, '0'         ; any more switches.
```

```
                putc
                mov         al, ' '
                putc

                mov         ah, 84h
                mov         dh, 9               ;Read all 16 switches.
                int         15h
                putw

                print
                byte        " Pots: ",0
                mov         ax, 8403h           ;Read joystick pots.
                mov         dx, 200h            ;Read four pots.
                int         15h
                puth
                mov         al, ' '
                putc
                mov         al, ah
                puth
                mov         al, ' '
                putc

                mov         ah, 84h
                mov         dx, 503h            ;Raw read, pot 3.
                int         15h
                putw

                putcr
                mov         ah, 1               ;Repeat until key press.
                int         16h
                je          MainLoop1
                getc



Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment     para stack 'stack'
stk             byte        1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment     para public 'zzzzzz'
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

24.6 An SGDI Driver for the CH Products' Flight Stick Pro™

The CH Product's FlightStick Pro joystick is a good example of a specialized product for which the SGDI driver is a perfect solution. The FlightStick Pro provides three pots and five switches, the fifth switch being a special five-position *cooley switch*. Although the pots on the FlightStick Pro map to three of the analog inputs on the standard game adapter card (pots zero, one, and three), there are insufficient digital inputs to handle the eight inputs necessary for the FlightStick Pro's four buttons and cooley switch.

The FlightStick Pro (FSP) uses some electronic circuitry to map these eight switch positions to four input bits. To do so, they place one restriction on the use of the FSP switches – you can only press one of them at a time. If you hold down two or more switches at the same time, the FSP hardware selects one of the switches and reports that value; it ignores the other switches until you release the button. Since only one switch can be read at a time, the FSP hardware generates a four bit value that determines the current state of the switches. It returns these four bits as the switch values on the standard game adapter card. The following table lists the values for each of the switches:

Table 88: FlightStick Pro Switch Return Values

Value (binary)	Priority	Switch Position
0000	Highest	Up position on the cooley switch.
0100	7	Right position on the cooley switch.
1000	6	Down position on the cooley switch.
1100	5	Left position on the cooley switch.
1110	4	Trigger on the joystick.
1101	3	Leftmost button on the joystick.
1011	2	Rightmost button on the joystick.
0111	Lowest	Middle button on the joystick.
1111		No buttons currently down.

Note that the buttons look just like a single button press. The cooley switch positions contain a position value in bits six and seven; bits four and five always contain zero when the cooley switch is active.

The SGDI driver for the FlightStick Pro is very similar to the standard game adapter card SGDI driver. Since the FlightStick Pro only provides three pots, this code doesn't bother trying to read pot 2 (which is non-existent). Of course, the switches on the FlightStick Pro are quite a bit different than those on standard joysticks, so the FSP SGDI driver maps the FPS switches to eight of the SGDI *logical* switches. By reading switches zero through seven, you can test the following conditions on the FSP:

Table 89: Flight Stick Pro SGDI Switch Mapping

This SGDI Switch number:	Maps to this FSP Switch:
0	Trigger on joystick.
1	Left button on joystick.
2	Middle button on joystick.
3	Right button on joystick.
4	Cooley up position.
5	Cooley left position.
6	Cooley right position.
7	Cooley down position.

The FSP SGDI driver contains one other novel feature, it will allow the user to swap the functions of the left and right switches on the joystick. Many games often assign important functions to the trigger and left button since they are easiest to press (right handed players can easily press the left button with their thumb). By typing "LEFT" on the command line, the FSP SGDI driver will swap the functions of the left and right buttons so left handed players can easily activate this function with their thumb as well.

The following code provides the complete listing for the FSPSGDI driver. Note that you can use the same test program from the previous section to test this driver.

```
                .286
                page        58, 132
                name        FSPSGDI
                title       FSPSGDI (CH Products Standard Game Device Interface).

; FSPSGDI.EXE
```

```
;
;       Usage:
;            FSPSDGI    {LEFT}
;
; This program loads a TSR which patches INT 15 so arbitrary game programs
; can read the CH Products FlightStick Pro joystick in a portable fashion.


wp           equ        <word ptr>
byp          equ        <byte ptr>



; We need to load cseg in memory before any other segments!

cseg         segment    para public 'code'
cseg         ends


; Initialization code, which we do not need except upon initial load,
; goes in the following segment:

Initialize   segment    para public 'INIT'
Initialize   ends

; UCR Standard Library routines which get dumped later on.

             .xlist
             include       stdlib.a
             includelib stdlib.lib
             .list

sseg         segment    para stack 'stack'
sseg         ends

zzzzzzseg    segment    para public 'zzzzzzseg'
zzzzzzseg    ends



CSEG         segment    para public 'CODE'
             assume     cs:cseg, ds:nothing

Int15Vect    dword      0

PSP          word       ?

; Port addresses for a typical joystick card:

JoyPort      equ        201h
JoyTrigger   equ        201h


CurrentReading word     0

Pot          struc
PotMask      byte       0                ;Pot mask for hardware.
DidCal       byte       0                ;Is this pot calibrated?
min          word       5000             ;Minimum pot value
max          word       0                ;Max pot value
center       word       0                ;Pot value in the middle
Pot          ends

Pot0         Pot        <1>
Pot1         Pot        <2>
Pot3         Pot        <8>


; SwapButtons- 0 if we should use normal flightstick pro buttons,
;              1 if we should swap the left and right buttons.

SwapButtons  byte       0

; SwBits- the four bit input value from the Flightstick Pro selects one
```

```
;       of the following bit patterns for a given switch position.

SwBits      byte    10h         ;Sw4
            byte    0           ;NA
            byte    0           ;NA
            byte    0           ;NA
            byte    40h         ;Sw6
            byte    0           ;NA
            byte    0           ;NA
            byte    4           ;Sw 2

            byte    80h         ;Sw 7
            byte    0           ;NA
            byte    0           ;NA
            byte    8           ;Sw 3
            byte    20h         ;Sw 5
            byte    2           ;Sw 1
            byte    1           ;Sw 0
            byte    0           ;NA

SwBitsL     byte    10h         ;Sw4
            byte    0           ;NA
            byte    0           ;NA
            byte    0           ;NA
            byte    40h         ;Sw6
            byte    0           ;NA
            byte    0           ;NA
            byte    4           ;Sw 2

            byte    80h         ;Sw 7
            byte    0           ;NA
            byte    0           ;NA
            byte    2           ;Sw 3
            byte    20h         ;Sw 5
            byte    8           ;Sw 1
            byte    1           ;Sw 0
            byte    0           ;NA


; The IDstring address gets passed back to the caller on a testpresence
; call. The four bytes before the IDstring must contain the serial number
; and current driver number.

SerialNumber byte   0,0,0
IDNumber    byte    0
IDString    byte    "CH Products:Flightstick Pro",0
            byte    "Written by Randall Hyde",0


;============================================================================
;
; ReadPots-   AH contains a bit mask to determine which pots we should read.
;             Bit 0 is one if we should read pot 0, bit 1 is one if we should
;             read pot 1, bit 3 is one if we should read pot 3. All other bits
;             will be zero.
;
;       This code returns the pot values in SI, BX, BP, and DI for Pot 0, 1,
;       2, & 3.
;

ReadPots    proc    near
            sub     bp, bp
            mov     si, bp
            mov     di, bp
            mov     bx, bp

; Wait for pots to finish any past junk:

            mov     dx, JoyPort
            out     dx, al          ;Trigger pots
            mov     cx, 400h
Wait4Pots:  in      al, dx
            and     al, 0Fh
```

```
                loopnz      Wait4Pots

; Okay, read the pots:

                mov         dx, JoyTrigger
                out         dx, al              ;Trigger pots
                mov         dx, JoyPort
                mov         cx, 8000h           ;Don't let this go on forever.
PotReadLoop:    in          al, dx
                and         al, ah
                jz          PotReadDone
                shr         al, 1
                adc         si, 0
                shr         al, 1
                adc         bp, 0
                shr         al, 2
                adc         di, 0
                loop        PotReadLoop
PotReadDone:
                ret
ReadPots        endp

;-----------------------------------------------------------------------------
;
; Normalize-    BX contains a pointer to a pot structure, AX contains
;               a pot value. Normalize that value according to the
;               calibrated pot.
;
; Note: DS must point at cseg before calling this routine.


                assume      ds:cseg
Normalize       proc        near
                push        cx

; Sanity check to make sure the calibration process went okay.

                cmp         [bx].Pot.DidCal, 0
                je          BadNorm
                mov         dx, [bx].Pot.Center
                cmp         dx, [bx].Pot.Min
                jbe         BadNorm
                cmp         dx, [bx].Pot.Max
                jae         BadNorm

; Clip the value if it is out of range.

                cmp         ax, [bx].Pot.Min
                ja          MinOkay
                mov         ax, [bx].Pot.Min
MinOkay:

                cmp         ax, [bx].Pot.Max
                jb          MaxOkay
                mov         ax, [bx].Pot.Max
MaxOkay:

; Scale this guy around the center:

                cmp         ax, [bx].Pot.Center
                jb          Lower128

; Scale in the range 128..255 here:

                sub         ax, [bx].Pot.Center
                mov         dl, ah              ;Multiply by 128
                mov         ah, al
                mov         dh, 0
                mov         al, dh
                shr         dl, 1
                rcr         ax, 1
                mov         cx, [bx].Pot.Max
                sub         cx, [bx].Pot.Center
                jz          BadNorm             ;Prevent division by zero.
```

```
                div     cx              ;Compute normalized value.
                add     ax, 128         ;Scale to range 128..255.
                cmp     ah, 0
                je      NormDone
                mov     ax, 0ffh        ;Result must fit in 8 bits!
                jmp     NormDone

; Scale in the range 0..127 here:

Lower128:       sub     ax, [bx].Pot.Min
                mov     dl, ah          ;Multiply by 128
                mov     ah, al
                mov     dh, 0
                mov     al, dh
                shr     dl, 1
                rcr     ax, 1
                mov     cx, [bx].Pot.Center
                sub     cx, [bx].Pot.Min
                jz      BadNorm
                div     cx              ;Compute normalized value.
                cmp     ah, 0
                je      NormDone
                mov     ax, 0ffh        ;Result must fit in 8 bits!
                jmp     NormDone

BadNorm:        sub     ax, ax
NormDone:       pop     cx
                ret
Normalize       endp
                assume  ds:nothing

;============================================================================
; INT 15h handler functions.
;============================================================================
;
; Although these are defined as near procs, they are not really procedures.
; The MyInt15 code jumps to each of these with BX, a far return address, and
; the flags sitting on the stack. Each of these routines must handle the
; stack appropriately.
;
;----------------------------------------------------------------------------
; BIOS- Handles the two BIOS calls, DL=0 to read the switches, DL=1 to
;       read the pots. For the BIOS routines, we'll ignore the cooley
;       switch (the hat) and simply read the other four switches.

BIOS            proc    near
                cmp     dl, 1           ;See if switch or pot routine.
                jb      Read4Sw
                je      ReadBIOSPots
                pop     bx
                jmp     cs:Int15Vect    ;Let someone else handle it!

Read4Sw:        push    dx
                mov     dx, JoyPort
                in      al, dx
                shr     al, 4
                mov     bl, al
                mov     bh, 0
                cmp     cs:SwapButtons, 0
                je      DoLeft2
                mov     al, cs:SwBitsL[bx]
                jmp     SBDone

DoLeft2:        mov     al, cs:SwBits[bx]
SBDone:         rol     al, 4           ;Put Sw0..3 in upper bits and make
                not     al              ; 0=switch down, just like game card.
                pop     dx
                pop     bx
                iret

ReadBIOSPots:   pop     bx              ;Return a value in BX!
                push    si
                push    di
                push    bp
```

```
                mov     ah, 0bh
                call    ReadPots
                mov     ax, si
                mov     bx, bp
                mov     dx, di
                sub     cx, cx
                pop     bp
                pop     di
                pop     si
                iret
BIOS            endp


;-----------------------------------------------------------------------------
;
; ReadPot-      On entry, DL contains a pot number to read.
;               Read and normalize that pot and return the result in AL.

                assume  ds:cseg
ReadPot         proc    near
;;;;;;;;;;      push    bx                  ;Already on stack.
                push    ds
                push    cx
                push    dx
                push    si
                push    di
                push    bp

                mov     bx, cseg
                mov     ds, bx

                cmp     dl, 0
                jne     Try1
                mov     ah, Pot0.PotMask
                call    ReadPots
                lea     bx, Pot0
                mov     ax, si
                call    Normalize
                jmp     GotPot

Try1:           cmp     dl, 1
                jne     Try3
                mov     ah, Pot1.PotMask
                call    ReadPots
                lea     bx, Pot1
                mov     ax, bp
                call    Normalize
                jmp     GotPot

Try3:           cmp     dl, 3
                jne     BadPot
                mov     ah, Pot3.PotMask
                call    ReadPots
                lea     bx, Pot3
                mov     ax, di
                call    Normalize
                jmp     GotPot

BadPot:         sub     ax, ax              ;Question: Should we pass this on
                                            ; or just return zero?
GotPot:         pop     bp
                pop     di
                pop     si
                pop     dx
                pop     cx
                pop     ds
                pop     bx
                iret
ReadPot         endp
                assume  ds:nothing


;-----------------------------------------------------------------------------
;
```

```
; ReadRaw-    On entry, DL contains a pot number to read.
;             Read that pot and return the unnormalized result in AL.

            assume  ds:cseg
ReadRaw     proc    near
;;;;;;;;;;  push    bx              ;Already on stack.
            push    ds
            push    cx
            push    dx
            push    si
            push    di
            push    bp

            mov     bx, cseg
            mov     ds, bx

            cmp     dl, 0
            jne     Try1
            mov     ah, Pot0.PotMask
            call    ReadPots
            mov     ax, si
            jmp     GotPot

Try1:       cmp     dl, 1
            jne     Try3
            mov     ah, Pot1.PotMask
            call    ReadPots
            mov     ax, bp
            jmp     GotPot

Try3:       cmp     dl, 3
            jne     BadPot
            mov     ah, Pot3.PotMask
            call    ReadPots
            mov     ax, di
            jmp     GotPot

BadPot:     sub     ax, ax          ;Just return zero.
GotPot:     pop     bp
            pop     di
            pop     si
            pop     dx
            pop     cx
            pop     ds
            pop     bx
            iret
ReadRaw     endp
            assume  ds:nothing


;-----------------------------------------------------------------------------
; Read4Pots-Reads pots zero, one, two, and three returning their
;           values in AL, AH, DL, and DH. Since the flightstick
;           Pro doesn't have a pot 2 installed, return zero for
;           that guy.

Read4Pots   proc    near
;;;;;;;;;;  push    bx              ;Already on stack
            push    ds
            push    cx
            push    si
            push    di
            push    bp

            mov     dx, cseg
            mov     ds, dx

            mov     ah, 0bh         ;Read pots 0, 1, and 3.
            call    ReadPots

            mov     ax, si
            lea     bx, Pot0
            call    Normalize
            mov     cl, al
```

```
                mov     ax, bp
                lea     bx, Pot1
                call    Normalize
                mov     ch, al

                mov     ax, di
                lea     bx, Pot3
                call    Normalize
                mov     dh, al          ;Pot 3 value.
                mov     ax, cx          ;Pots 0 and 1.
                mov     dl, 0           ;Pot 2 is non-existant.

                pop     bp
                pop     di
                pop     si
                pop     cx
                pop     ds
                pop     bx
                iret
Read4Pots       endp



;-----------------------------------------------------------------------------
; CalPot-       Calibrate the pot specified by DL. On entry, AL contains
;               the minimum pot value (it better be less than 256!), BX
;               contains the maximum pot value, and CX contains the centered
;               pot value.

                assume  ds:cseg
CalPot          proc    near
                pop     bx              ;Retrieve maximum value
                push    ds
                push    si
                mov     si, cseg
                mov     ds, si

; Sanity check on parameters, sort them in ascending order:

                mov     ah, 0
                cmp     bx, cx
                ja      GoodMax
                xchg    bx, cx
GoodMax:        cmp     ax, cx
                jb      GoodMin
                xchg    ax, cx
GoodMin:        cmp     cx, bx
                jb      GoodCenter
                xchg    cx, bx
GoodCenter:


; Okay, figure out who were supposed to calibrate:

                lea     si, Pot0
                cmp     dl, 1
                jb      DoCal
                lea     si, Pot1
                je      DoCal
                cmp     dl, 3
                jne     CalDone
                lea     si, Pot3

DoCal:          mov     [si].Pot.min, ax
                mov     [si].Pot.max, bx
                mov     [si].Pot.center, cx
                mov     [si].Pot.DidCal, 1
CalDone:        pop     si
                pop     ds
                iret
CalPot          endp
                assume  ds:nothing
```

```
;-----------------------------------------------------------------------------
; TestCal-      Just checks to see if the pot specified by DL has already
;               been calibrated.

                assume  ds:cseg
TestCal         proc    near
;;;;;;;;        push    bx              ;Already on stack
                push    ds
                mov     bx, cseg
                mov     ds, bx

                sub     ax, ax          ;Assume no calibration
                lea     bx, Pot0
                cmp     dl, 1
                jb      GetCal
                lea     bx, Pot1
                je      GetCal
                cmp     dl, 3
                jne     BadCal
                lea     bx, Pot3

GetCal:         mov     al, [bx].Pot.DidCal
                mov     ah, 0
BadCal:         pop     ds
                pop     bx
                iret
TestCal         endp
                assume  ds:nothing


;-----------------------------------------------------------------------------
;
; ReadSw-       Reads the switch whose switch number appears in DL.

SwTable         byte    11100000b, 11010000b, 01110000b, 10110000b
                byte    00000000b, 11000000b, 01000000b, 10000000b

SwTableL        byte    11100000b, 10110000b, 01110000b, 11010000b
                byte    00000000b, 11000000b, 01000000b, 10000000b

ReadSw          proc    near
;;;;;;;         push    bx              ;Already on stack
                mov     bl, dl          ;Save switch to read.
                mov     bh, 0
                mov     dx, JoyPort
                in      al, dx
                and     al, 0f0h
                cmp     cs:SwapButtons, 0
                je      DoLeft0
                cmp     al, cs:SwTableL[bx]
                jne     NotDown
                jmp     IsDown

DoLeft0:        cmp     al, cs:SwTable[bx]
                jne     NotDown

IsDown:         mov     ax, 1
                pop     bx
                iret

NotDown:        sub     ax, ax
                pop     bx
                iret
ReadSw          endp


;-----------------------------------------------------------------------------
;
; Read16Sw-     Reads all eight switches and returns their values in AX.

Read16Sw        proc    near
;;;;;;;;        push    bx              ;Already on stack
```

```
                mov     ah, 0           ;Switches 8-15 are non-existant.
                mov     dx, JoyPort
                in      al, dx
                shr     al, 4
                mov     bl, al
                mov     bh, 0
                cmp     cs:SwapButtons, 0
                je      DoLeft1
                mov     al, cs:SwBitsL[bx]
                jmp     R8Done

DoLeft1:        mov     al, cs:SwBits[bx]
R8Done:         pop     bx
                iret
Read16Sw        endp


;*****************************************************************************
;
; MyInt15-      Patch for the BIOS INT 15 routine to control reading the
;               joystick.

MyInt15         proc    far
                push    bx
                cmp     ah, 84h         ;Joystick code?
                je      DoJoystick
OtherInt15:     pop     bx
                jmp     cs:Int15Vect

DoJoystick:     mov     bh, 0
                mov     bl, dh
                cmp     bl, 80h
                jae     VendorCalls
                cmp     bx, JmpSize
                jae     OtherInt15
                shl     bx, 1
                jmp     wp cs:jmptable[bx]

jmptable        word    BIOS
                word    ReadPot, Read4Pots, CalPot, TestCal
                word    ReadRaw, OtherInt15, OtherInt15
                word    ReadSw, Read16Sw
JmpSize         =       ($-jmptable)/2


; Handle vendor specific calls here.

VendorCalls:    je      RemoveDriver
                cmp     bl, 81h
                je      TestPresence
                pop     bx
                jmp     cs:Int15Vect

; TestPresence- Returns zero in AX and a pointer to the ID string in ES:BX

TestPresence:   pop     bx              ;Get old value off stack.
                sub     ax, ax
                mov     bx, cseg
                mov     es, bx
                lea     bx, IDString
                iret

; RemoveDriver-If there are no other drivers loaded after this one in
;               memory, disconnect it and remove it from memory.

RemoveDriver:
                push    ds
                push    es
                push    ax
                push    dx

                mov     dx, cseg
                mov     ds, dx
```

```
; See if we're the last routine patched into INT 15h

                mov     ax, 3515h
                int     21h
                cmp     bx, offset MyInt15
                jne     CantRemove
                mov     bx, es
                cmp     bx, wp seg MyInt15
                jne     CantRemove

                mov     ax, PSP         ;Free the memory we're in
                mov     es, ax
                push    es
                mov     ax, es:[2ch]    ;First, free env block.
                mov     es, ax
                mov     ah, 49h
                int     21h
;
                pop     es              ;Now free program space.
                mov     ah, 49h
                int     21h

                lds     dx, Int15Vect   ;Restore previous int vect.
                mov     ax, 2515h
                int     21h

CantRemove:     pop     dx
                pop     ax
                pop     es
                pop     ds
                pop     bx
                iret
MyInt15         endp
cseg            ends


; The following segment is tossed when this code goes resident.

Initialize      segment para public 'INIT'
                assume  cs:Initialize, ds:cseg
Main            proc
                mov     ax, cseg        ;Get ptr to vars segment
                mov     es, ax
                mov     es:PSP, ds      ;Save PSP value away
                mov     ds, ax

                mov     ax, zzzzzzseg
                mov     es, ax
                mov     cx, 100h
                meminit2

                print
                byte    "Standard Game Device Interface driver",cr,lf
                byte    "CH Products Flightstick Pro",cr,lf
                byte    "Written by Randall Hyde",cr,lf
                byte    cr,lf
                byte    "'FSPSGDI LEFT' swaps the left and right buttons for "
                byte    "left handed players",cr,lf
                byte    "'FSPSGDI REMOVE' removes the driver from memory"
                byte    cr, lf, lf
                byte    0

                mov     ax, 1
                argv                    ;If no parameters, empty str.
                stricmpl
                byte    "LEFT",0
                jne     NoLEFT
                mov     SwapButtons, 1
                print
                byte    "Left and right buttons swapped",cr,lf,0
                jmp     SwappedLeft

NoLEFT:         stricmpl
```

```
                byte    "REMOVE",0
                jne     NoRmv
                mov     dh, 81h
                mov     ax, 84ffh
                int     15h             ;See if we're already loaded.
                test    ax, ax          ;Get a zero back?
                jz      Installed
                print
                byte    "SGDI driver is not present in memory, REMOVE "
                byte    "command ignored.",cr,lf,0
                mov     ax, 4c01h;Exit to DOS.
                int     21h

Installed:      mov     ax, 8400h
                mov     dh, 80h             ;Remove call
                int     15h
                mov     ax, 8400h
                mov     dh, 81h             ;TestPresence call
                int     15h
                cmp     ax, 0
                je      NotRemoved
                print
                byte    "Successfully removed SGDI driver from memory."
                byte    cr,lf,0
                mov     ax, 4c01h       ;Exit to DOS.
                int     21h

NotRemoved:     print
                byte    "SGDI driver is still present in memory.",cr,lf,0
                mov     ax, 4c01h;Exit to DOS.
                int     21h


NoRmv:


; Okay, Patch INT 15 and go TSR at this point.

SwappedLeft:    mov     ax, 3515h
                int     21h
                mov     wp Int15Vect, bx
                mov     wp Int15Vect+2, es

                mov     dx, cseg
                mov     ds, dx
                mov     dx, offset MyInt15
                mov     ax, 2515h
                int     21h

                mov     dx, cseg
                mov     ds, dx
                mov     dx, seg Initialize
                sub     dx, ds:psp
                add     dx, 2
                mov     ax, 3100h           ;Do TSR
                int     21h
Main            endp

Initialize      ends

sseg            segment para stack 'stack'
                word    128 dup (0)
endstk          word    ?
sseg            ends

zzzzzzseg       segment para public 'zzzzzzseg'
                byte    16 dup (0)
zzzzzzseg       ends
                end     Main
```

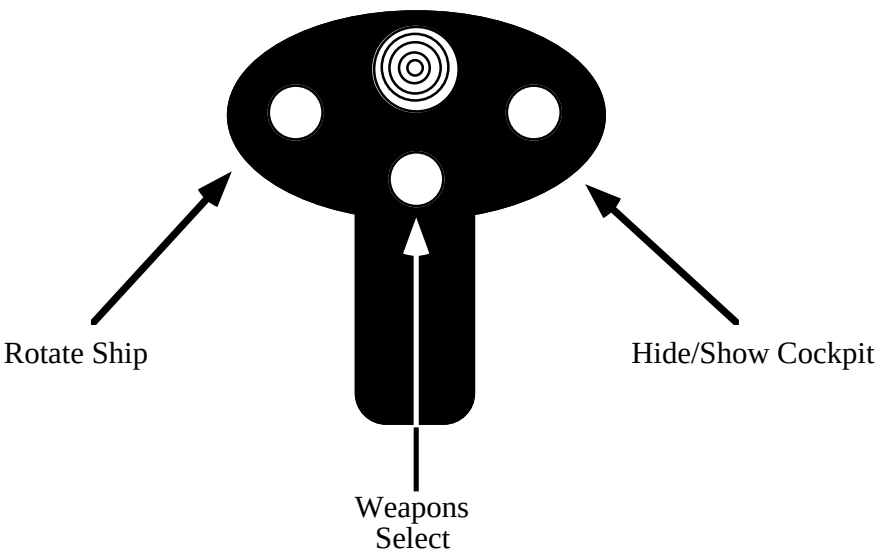
24.7 Patching Existing Games

Maybe you're not quite ready to write the next million dollar game. Perhaps you'd like to get a little more enjoyment out of the games you already own. Well, this section will provide a practical application of a semiresident program that patches the Lucas Arts' XWing (Star Wars simulation) game. This program patches the XWing game to take advantage of the special features found on the CH Products' FlightStick Pro. In particular, it lets you use the throttle pot on the FSP to control the speed of the spacecraft. It also lets you program each of the buttons with up to four strings of eight characters each.

To describe how you can patch an existing game, a short description of how this patch was developed is in order. The FSPXW patch was developed by using the Soft-ICE™debugging tool. This program lets you set a breakpoint whenever an 80386 or later processor accesses a specific I/O port[8]. Setting a breakpoint at I/O address 201h while running the xwing.exe file stopped the XWing program when it decided to read the analog and switch inputs. Disassembly of the surrounding code produced complete joystick and button read routines. After locating these routines, it was easy enough to write a program to search through memory for the code and patch in jumps to code in the FSPXW patch program.

Note that the original joystick code inside XWing works perfectly fine with the FPS. The only reason for patching into the joystick code is so our code can read the throttle every how and then and take appropriate action.

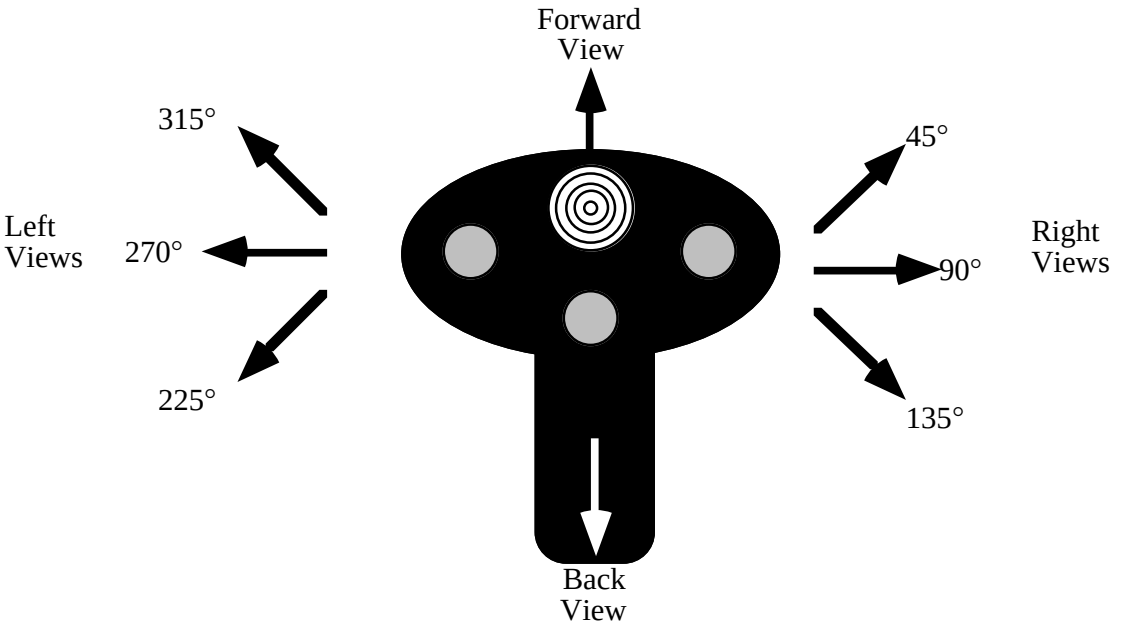
The button routines were another story altogether. The FSPXW patch needs to take control of XWing's button routines because the user of FSPXW might want to redefine a button recognized by XWing for some other purpose. Therefore, whenever XWing calls its button routine, control transfers to the button routine inside FSPXW that decides whether to pass real button information back to XWing or to fake buttons in the up position because those buttons are redefined to other functions. By default (unless you change the source code, the buttons have the following programming:



The programming of the cooley switch demonstrates an interesting feature of the FSPXW patch: you can program up to four different strings on each button. The first time you press a button, FSPXW emits the first string, the second time you press a button it emits the second string, then the third, and finally the fourth. If the string is empty, the FSPXW string skips it. The FSPXW patch uses the cooley switch to select the cockpit views. Pressing the cooley switch forward displays the forward view. Pulling the cooley switch backwards presents the rear view. However, the XWing game provides *three* left and right views. Pushing the cooley switch to the left or right once displays the 45 degree view. Pressing it a second time presents

8. This feature is not specific to Soft-ICE, many 80386 debuggers will let you do this.

the 90 degree view. Pressing it to the left or right a third time provides the 135 degree view. The following diagram shows the default programming on the cooley switch:



One word of caution concerning this patch: it only works with the basic XWing game. It does not support the add-on modules (Imperial Pursuit, B-Wing, Tie Fighter, etc.). Furthermore, this patch assumes that the basic XWing code has not changed over the years. It could be that a recent release of the XWing game uses new joystick routines and the code associated with this application will not be able to locate or patch those new routines. This patch will detect such a situation and will not patch XWing if this is the case. You must have sufficient free RAM for this patch, XWing, and anything else you have loaded into memory at the same time (the exact amount of RAM XWing needs depends upon the features you've installed, a fully installed system requires slightly more than 610K free).

Without further ado, here's the FSPXW code:

```
                .286
                page        58, 132
                name        FSPXW
                title       FSPXW (Flightstick Pro driver for XWING).
                subttl      Copyright (C) 1994 Randall Hyde.


; FSPXW.EXE
;
;       Usage:
;             FSPXW
;
; This program executes the XWING.EXE program and patches it to use the
; Flightstick Pro.


byp             textequ     <byte ptr>
wp              textequ     <word ptr>


cseg            segment para public 'CODE'
cseg            ends

sseg            segment     para stack 'STACK'
sseg            ends

zzzzzzseg       segment     para public 'zzzzzzseg'
zzzzzzseg       ends
```

```
                include       stdlib.a
                includelib stdlib.lib
                matchfuncs

                ifndef     debug
Installation    segment    para public 'Install'
Installation    ends
                endif

CSEG            segment    para public 'CODE'
                assume     cs:cseg, ds:nothing

; Timer interrupt vector

Int1CVect       dword      ?


; PSP-   Program Segment Prefix. Needed to free up memory before running
;        the real application program.

PSP             word       0



; Program Loading data structures (for DOS).

ExecStruct      word       0                ;Use parent's Environment blk.
                dword      CmdLine          ;For the cmd ln parms.
                dword      DfltFCB
                dword      DfltFCB
LoadSSSP        dword      ?
LoadCSIP        dword      ?
PgmName         dword      Pgm


; Variables for the throttle pot.
; LastThrottle contains the character last sent (so we only send one copy).
; ThrtlCntDn counts the number of times the throttle routine gets called.

LastThrottle    byte       0
ThrtlCntDn      byte       10


; Button Mask- Used to mask out the programmed buttons when the game
; reads the real buttons.

ButtonMask      byte       0f0h


; The following variables allow the user to reprogram the buttons.

KeyRdf          struct
Ptrs            word       ?             ;The PTRx fields point at the
ptr2            word       ?             ; four possible strings of 8 chars
ptr3            word       ?             ; each. Each button press cycles
ptr4            word       ?             ; through these strings.
Index           word       ?             ;Index to next string to output.
Cnt             word       ?
Pgmd            word       ?             ;Flag = 0 if not redefined.
KeyRdf          ends


; Left codes are output if the cooley switch is pressed to the left.
; Note that the strings ares actually zero terminated strings of words.

Left            KeyRdf     <Left1, Left2, Left3, Left4, 0, 6, 1>
Left1           word       '7', 0
Left2           word       '4', 0
Left3           word       '1', 0
Left4           word       0

; Right codes are output if the cooley switch is pressed to the Right.
```

```
Right           KeyRdf      <Right1, Right2, Right3, Right4, 0, 6, 1>
Right1          word        '9', 0
Right2          word        '6', 0
Right3          word        '3', 0
Right4          word        0

; Up codes are output if the cooley switch is pressed Up.

Up              KeyRdf      <Up1, Up2, Up3, Up4, 0, 2, 1>
Up1             word        '8', 0
Up2             word        0
Up3             word        0
Up4             word        0

; DownKey codes are output if the cooley switch is pressed Down.

Down            KeyRdf      <Down1, Down2, Down3, Down4, 0, 2, 1>
Down1           word        '2', 0
Down2           word        0
Down3           word        0
Down4           word        0

; Sw0 codes are output if the user pulls the trigger.(This switch is not
; redefined.)

Sw0             KeyRdf      <Sw01, Sw02, Sw03, Sw04, 0, 0, 0>
Sw01            word        0
Sw02            word        0
Sw03            word        0
Sw04            word        0


; Sw1 codes are output if the user presses Sw1 (the left button
; if the user hasn't swapped the left and right buttons). Not Redefined.

Sw1             KeyRdf      <Sw11, Sw12, Sw13, Sw14, 0, 0, 0>
Sw11            word        0
Sw12            word        0
Sw13            word        0
Sw14            word        0

; Sw2 codes are output if the user presses Sw2 (the middle button).

Sw2             KeyRdf      <Sw21, Sw22, Sw23, Sw24, 0, 2, 1>
Sw21            word        'w', 0
Sw22            word        0
Sw23            word        0
Sw24            word        0

; Sw3 codes are output if the user presses Sw3 (the right button
; if the user hasn't swapped the left and right buttons).

Sw3             KeyRdf      <Sw31, Sw32, Sw33, Sw34, 0, 0, 0>
Sw31            word        0
Sw32            word        0
Sw33            word        0
Sw34            word        0

; Switch status buttons:

CurSw           byte        0
LastSw          byte        0


;*****************************************************************************
; FSPXW patch begins here. This is the memory resident part. Only put code
; which which has to be present at run-time or needs to be resident after
; freeing up memory.
;*****************************************************************************

Main            proc
                mov         cs:PSP, ds
                mov         ax, cseg        ;Get ptr to vars segment
                mov         ds, ax
```

```
; Get the current INT 1Ch interrupt vector:

                mov     ax, 351ch
                int     21h
                mov     wp Int1CVect, bx
                mov     wp Int1CVect+2, es

; The following call to MEMINIT assumes no error occurs. If it does,
; we're hosed anyway.

                mov     ax, zzzzzzseg
                mov     es, ax
                mov     cx, 1024/16
                meminit2


; Do some initialization before running the game. These are calls to the
; initialization code which gets dumped before actually running XWING.

                call    far ptr ChkBIOS15
                call    far ptr Identify
                call    far ptr Calibrate

; If any switches were programmed, remove those switches from the
; ButtonMask:

                mov     al, 0f0h        ;Assume all buttons are okay.
                cmp     sw0.pgmd, 0
                je      Sw0NotPgmd
                and     al, 0e0h        ;Remove sw0 from contention.
Sw0NotPgmd:

                cmp     sw1.pgmd, 0
                je      Sw1NotPgmd
                and     al, 0d0h        ;Remove Sw1 from contention.
Sw1NotPgmd:

                cmp     sw2.pgmd, 0
                je      Sw2NotPgmd
                and     al, 0b0h        ;Remove Sw2 from contention.
Sw2NotPgmd:

                cmp     sw3.pgmd, 0
                je      Sw3NotPgmd
                and     al, 070h        ;Remove Sw3 from contention.
Sw3NotPgmd:
                mov     ButtonMask, al  ;Save result as button mask

; Now, free up memory from ZZZZZZSEG on to make room for XWING.
; Note: Absolutely no calls to UCR Standard Library routines from
; this point forward! (ExitPgm is okay, it's just a macro which calls DOS.)
; Note that after the execution of this code, none of the code & data
; from zzzzzzseg on is valid.

                mov     bx, zzzzzzseg
                sub     bx, PSP
                inc     bx
                mov     es, PSP
                mov     ah, 4ah
                int     21h
                jnc     GoodRealloc
                print
                byte    "Memory allocation error."
                byte    cr,lf,0
                jmp     Quit

GoodRealloc:

; Now load the XWING program into memory:

                mov     bx, seg ExecStruct
                mov     es, bx
```

```
                mov     bx, offset ExecStruc ;Ptr to program record.
                lds     dx, PgmName
                mov     ax, 4b01h       ;Load, do not exec, pgm
                int     21h
                jc      Quit            ;If error loading file.

; Search for the joystick code in memory:

                mov     si, zzzzzzseg
                mov     ds, si
                xor     si, si

                mov     di, cs
                mov     es, di
                mov     di, offset JoyStickCode
                mov     cx, JoyLength
                call    FindCode
                jc      Quit            ;If didn't find joystick code.


; Patch the XWING joystick code here

                mov     byp ds:[si], 09ah               ;Far call
                mov     wp ds:[si+1], offset ReadGame
                mov     wp ds:[si+3], cs

; Find the Button code here.

                mov     si, zzzzzzseg
                mov     ds, si
                xor     si, si

                mov     di, cs
                mov     es, di
                mov     di, offset ReadSwCode
                mov     cx, ButtonLength
                call    FindCode
                jc      Quit

; Patch the button code here.

                mov     byp ds:[si], 9ah
                mov     wp ds:[si+1], offset ReadButtons
                mov     wp ds:[si+3], cs
                mov     byp ds:[si+5], 90h              ;NOP.


; Patch in our timer interrupt handler:

                mov     ax, 251ch
                mov     dx, seg MyInt1C
                mov     ds, dx
                mov     dx, offset MyInt1C
                int     21h


; Okay, start the XWING.EXE program running

                mov     ah, 62h         ;Get PSP
                int     21h
                mov     ds, bx
                mov     es, bx
                mov     wp ds:[10], offset Quit
                mov     wp ds:[12], cs
                mov     ss, wp cseg:LoadSSSP+2
                mov     sp, wp cseg:LoadSSSP
                jmp     dword ptr cseg:LoadCSIP


Quit:           lds     dx, cs:Int1CVect  ;Restore timer vector.
                mov     ax, 251ch
                int     21h
                ExitPgm
```

```
Main            endp

;*****************************************************************************
;
; ReadGame-     This routine gets called whenever XWing reads the joystick.
;               On every 10th call it will read the throttle pot and send
;               appropriate characters to the type ahead buffer, if
;               necessary.

                assume  ds:nothing
ReadGame        proc    far
                dec     cs:ThrtlCntDn     ;Only do this each 10th time
                jne     SkipThrottle      ; XWING calls the joystick
                mov     cs:ThrtlCntDn, 10 ; routine.

                push    ax
                push    bx                ;No need to save bp, dx, or cx as
                push    di                ; XWING preserves these.

                mov     ah, 84h
                mov     dx, 103h          ;Read the throttle pot
                int     15h

; Convert the value returned by the pot routine into the four characters
; 0..63:"\", 64..127:"[", 128..191:"]", 192..255:<bs>, to denote zero, 1/3,
; 2/3, and full power, respectively.

                mov     dl, al
                mov     ax, "\"           ;Zero power
                cmp     dl, 192
                jae     SetPower
                mov     ax, "["           ;1/3 power.
                cmp     dl, 128
                jae     SetPower
                mov     ax, "]"           ;2/3 power.
                cmp     dl, 64
                jae     SetPower
                mov     ax, 8             ;BS, full power.
SetPower:       cmp     al, cs:LastThrottle
                je      SkipPIB
                mov     cs:LastThrottle, al
                call    PutInBuffer

SkipPIB:        pop     di
                pop     bx
                pop     ax
SkipThrottle:   neg     bx            ;XWING returns data in these registers.
                neg     di            ;We patched the NEG and STI instrs
                sti                   ; so do that here.
                ret
ReadGame        endp

                assume  ds:nothing
ReadButtons     proc    far
                mov     ah, 84h
                mov     dx, 0
                int     15h
                not     al
                and     al, ButtonMask    ;Turn off pgmd buttons.
                ret
ReadButtons     endp



; MyInt1C- Called every 1/18th second. Reads switches and decides if it
; should shove some characters into the type ahead buffer.

                assume  ds:cseg
MyInt1c         proc    far
                push    ds
                push    ax
                push    bx
                push    dx
                mov     ax, cseg
```

```
                mov     ds, ax

                mov     al, CurSw
                mov     LastSw, al

                mov     dx, 900h        ;Read the 8 switches.
                mov     ah, 84h
                int     15h

                mov     CurSw, al
                xor     al, LastSw      ;See if any changes
                jz      NoChanges
                and     al, CurSw       ;See if sw just went down.
                jz      NoChanges


; If a switch has just gone down, output an appropriate set of scan codes
; for it, if that key is active. Note that pressing *any* key will reset
; all the other key indexes.

                test    al, 1           ;See if Sw0 (trigger) was pulled.
                jz      NoSw0
                cmp     Sw0.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Left.Index, ax  ;Reset the key indexes for all keys
                mov     Right.Index, ax ; except SW0.
                mov     Up.Index, ax
                mov     Down.Index, ax
                mov     Sw1.Index, ax
                mov     Sw2.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Sw0.Index
                mov     ax, Sw0.Index
                mov     bx, Sw0.Ptrs[bx]
                add     ax, 2
                cmp     ax, Sw0.Cnt
                jb      SetSw0
                mov     ax, 0
SetSw0:         mov     Sw0.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoSw0:          test    al, 2           ;See if Sw1 (left sw) was pressed.
                jz      NoSw1
                cmp     Sw1.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Left.Index, ax  ;Reset the key indexes for all keys
                mov     Right.Index, ax ; except Sw1.
                mov     Up.Index, ax
                mov     Down.Index, ax
                mov     Sw0.Index, ax
                mov     Sw2.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Sw1.Index
                mov     ax, Sw1.Index
                mov     bx, Sw1.Ptrs[bx]
                add     ax, 2
                cmp     ax, Sw1.Cnt
                jb      SetSw1
                mov     ax, 0
SetSw1:         mov     Sw1.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoSw1:          test    al, 4           ;See if Sw2 (middle sw) was pressed.
                jz      NoSw2
                cmp     Sw2.Pgmd, 0
                je      NoChanges
                mov     ax, 0
```

```
                mov     Left.Index, ax    ;Reset the key indexes for all keys
                mov     Right.Index, ax   ; except Sw2.
                mov     Up.Index, ax
                mov     Down.Index, ax
                mov     Sw0.Index, ax
                mov     Sw1.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Sw2.Index
                mov     ax, Sw2.Index
                mov     bx, Sw2.Ptrs[bx]
                add     ax, 2
                cmp     ax, Sw2.Cnt
                jb      SetSw2
                mov     ax, 0
SetSw2:         mov     Sw2.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoSw2:          test    al, 8             ;See if Sw3 (right sw) was pressed.
                jz      NoSw3
                cmp     Sw3.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Left.Index, ax    ;Reset the key indexes for all keys
                mov     Right.Index, ax   ; except Sw3.
                mov     Up.Index, ax
                mov     Down.Index, ax
                mov     Sw0.Index, ax
                mov     Sw1.Index, ax
                mov     Sw2.Index, ax
                mov     bx, Sw3.Index
                mov     ax, Sw3.Index
                mov     bx, Sw3.Ptrs[bx]
                add     ax, 2
                cmp     ax, Sw3.Cnt
                jb      SetSw3
                mov     ax, 0
SetSw3:         mov     Sw3.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoSw3:          test    al, 10h           ;See if Cooly was pressed upwards.
                jz      NoUp
                cmp     Up.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Right.Index, ax   ;Reset all but Up.
                mov     Left.Index, ax
                mov     Down.Index, ax
                mov     Sw0.Index, ax
                mov     Sw1.Index, ax
                mov     Sw2.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Up.Index
                mov     ax, Up.Index
                mov     bx, Up.Ptrs[bx]
                add     ax, 2
                cmp     ax, Up.Cnt
                jb      SetUp
                mov     ax, 0
SetUp:          mov     Up.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoUp:           test    al, 20h           ;See if Cooley was pressed left.
                jz      NoLeft
                cmp     Left.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Right.Index, ax   ;Reset all but Left.
                mov     Up.Index, ax
```

```
                mov     Down.Index, ax
                mov     Sw0.Index, ax
                mov     Sw1.Index, ax
                mov     Sw2.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Left.Index
                mov     ax, Left.Index
                mov     bx, Left.Ptrs[bx]
                add     ax, 2
                cmp     ax, Left.Cnt
                jb      SetLeft
                mov     ax, 0
SetLeft:        mov     Left.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoLeft:         test    al, 40h         ;See if Cooley was pressed Right
                jz      NoRight
                cmp     Right.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Left.Index, ax  ;Reset all but Right.
                mov     Up.Index, ax
                mov     Down.Index, ax
                mov     Sw0.Index, ax
                mov     Sw1.Index, ax
                mov     Sw2.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Right.Index
                mov     ax, Right.Index
                mov     bx, Right.Ptrs[bx]
                add     ax, 2
                cmp     ax, Right.Cnt
                jb      SetRight
                mov     ax, 0
SetRight:       mov     Right.Index, ax
                call    PutStrInBuf
                jmp     NoChanges


NoRight:        test    al, 80h         ;See if Cooly was pressed Downward.
                jz      NoChanges
                cmp     Down.Pgmd, 0
                je      NoChanges
                mov     ax, 0
                mov     Left.Index, ax  ;Reset all but Down.
                mov     Up.Index, ax
                mov     Right.Index, ax
                mov     Sw0.Index, ax
                mov     Sw1.Index, ax
                mov     Sw2.Index, ax
                mov     Sw3.Index, ax
                mov     bx, Down.Index
                mov     ax, Down.Index
                mov     bx, Down.Ptrs[bx]
                add     ax, 2
                cmp     ax, Down.Cnt
                jb      SetDown
                mov     ax, 0
SetDown:        mov     Down.Index, ax
                call    PutStrInBuf

NoChanges:      pop     dx
                pop     bx
                pop     ax
                pop     ds
                jmp     cs:Int1CVect
MyInt1c         endp
                assume  ds:nothing

; PutStrInBuf- BX points at a zero terminated string of words.
;              Output each word by calling PutInBuffer.
```

```
PutStrInBuf   proc      near
              push      ax
              push      bx
PutLoop:      mov       ax, [bx]
              test      ax, ax
              jz        PutDone
              call      PutInBuffer
              add       bx, 2
              jmp       PutLoop

PutDone:      pop       bx
              pop       ax
              ret
PutStrInBuf   endp

; PutInBuffer- Outputs character and scan code in AX to the type ahead
; buffer.

              assume    ds:nothing
KbdHead       equ       word ptr ds:[1ah]
KbdTail       equ       word ptr ds:[1ch]
KbdBuffer     equ       word ptr ds:[1eh]
EndKbd        equ       3eh
Buffer        equ       1eh

PutInBuffer   proc      near
              push      ds
              push      bx
              mov       bx, 40h
              mov       ds, bx
              pushf
              cli                           ;This is a critical region!
              mov       bx, KbdTail         ;Get ptr to end of type
              inc       bx                  ; ahead buffer and make room
              inc       bx                  ; for this character.
              cmp       bx, buffer+32       ;At physical end of buffer?
              jb        NoWrap
              mov       bx, buffer          ;Wrap back to 1eH if at end.
;
NoWrap:       cmp       bx, KbdHead         ;Buffer overrun?
              je        PIBDone
              xchg      KbdTail, bx         ;Set new, get old, ptrs.
              mov       ds:[bx], ax         ;Output AX to old location.
PIBDone:      popf                          ;Restore interrupts
              pop       bx
              pop       ds
              ret
PutInBuffer   endp



;*****************************************************************************
;
; FindCode: On entry, ES:DI points at some code in *this* program which
;         appears in the ATP game. DS:SI points at a block of memory
;         in the XWing game. FindCode searches through memory to find the
;         suspect piece of code and returns DS:SI pointing at the start of
;         that code. This code assumes that it *will* find the code!
;         It returns the carry clear if it finds it, set if it doesn't.

FindCode      proc      near
              push      ax
              push      bx
              push      dx

DoCmp:        mov       dx, 1000h
CmpLoop:      push      di                  ;Save ptr to compare code.
              push      si                  ;Save ptr to start of string.
              push      cx                  ;Save count.
         repe cmpsb
              pop       cx
              pop       si
              pop       di
              je        FoundCode
```

```
                inc     si
                dec     dx
                jne     CmpLoop
                sub     si, 1000h
                mov     ax, ds
                inc     ah
                mov     ds, ax
                cmp     ax, 9000h
                jb      DoCmp

                pop     dx
                pop     bx
                pop     ax
                stc
                ret

FoundCode:      pop     dx
                pop     bx
                pop     ax
                clc
                ret
FindCode        endp


;*****************************************************************************
;
; Joystick and button routines which appear in XWing game. This code is
; really data as the INT 21h patch code searches through memory for this code
; after loading a file from disk.


JoyStickCode    proc    near
                sti
                neg     bx
                neg     di
                pop     bp
                pop     dx
                pop     cx
                ret
                mov     bp, bx
                in      al, dx
                mov     bl, al
                not     al
                and     al, ah
                jnz     $+11h
                in      al, dx
JoyStickCode    endp
EndJSC:

JoyLength       =       EndJSC-JoyStickCode

ReadSwCode      proc
                mov     dx, 201h
                in      al, dx
                xor     al, 0ffh
                and     ax, 0f0h
ReadSwCode      endp
EndRSC:

ButtonLength    =       EndRSC-ReadSwCode

cseg            ends


Installation    segment

; Move these things here so they do not consume too much space in the
; resident part of the patch.

DfltFCB         byte    3," ",0,0,0,0,0
CmdLine         byte    2, " ", 0dh, 126 dup (" ")      ;Cmd line for program
Pgm             byte    "XWING.EXE",0
                byte    128 dup (?)                     ;For user's name
```

```
; ChkBIOS15- Checks to see if the INT 15 driver for FSPro is present in memory.

ChkBIOS15     proc      far
              mov       ah, 84h
              mov       dx, 8100h
              int       15h
              mov       di, bx
              strcmpl
              byte      "CH Products:Flightstick Pro",0
              jne       NoDriverLoaded
              ret

NoDriverLoaded:
              print
              byte      "CH Products SGDI driver for Flightstick Pro is not "
              byte      "loaded into memory.",cr,lf
              byte      "Please run FSPSGDI before running this program."
              byte      cr,lf,0
              exitpgm

ChkBIOS15     endp


;*****************************************************************************
;
; Identify-   Prints a sign-on message.

              assume    ds:nothing
Identify      proc      far

; Print a welcome string. Note that the string "VersionStr" will be
; modified by the "version.exe" program each time you assemble this code.

              print
              byte      cr,lf,lf
              byte      "X W I N G P A T C H",cr,lf
              byte      "CH Products Flightstick Pro",cr,lf
              byte      "Copyright 1994, Randall Hyde",cr,lf
              byte      lf
              byte      0

              ret
Identify      endp

;*****************************************************************************
;
; Calibrate the throttle down here:

              assume    ds:nothing
Calibrate     proc      far
              print
              byte      cr,lf,lf
              byte      "Calibration:",cr,lf,lf
              byte      "Move the throttle to one extreme and press any "
              byte      "button:",0

              call      Wait4Button
              mov       ah, 84h
              mov       dx, 1h
              int       15h
              push      dx                ;Save pot 3 reading.

              print
              byte      cr,lf
              byte      "Move the throttle to the other extreme and press "
              byte      "any button:",0

              call      Wait4Button
              mov       ah, 84h
              mov       dx, 1
              int       15h
              pop       bx
```

```
                mov     ax, dx
                cmp     ax, bx
                jb      RangeOkay
                xchg    ax, bx
RangeOkay:      mov     cx, bx          ;Compute a centered value.
                sub     cx, ax
                shr     cx, 1
                add     cx, ax
                mov     ah, 84h
                mov     dx, 303h        ;Calibrate pot three.
                int     15h
                ret
Calibrate       endp


Wait4Button     proc    near
                mov     ah, 84h         ;First, wait for all buttons
                mov     dx, 0           ; to be released.
                int     15h
                and     al, 0F0h
                cmp     al, 0F0h
                jne     Wait4Button

                mov     cx, 0
Delay:          loop    Delay

Wait4Press:     mov     ah, 1           ;Eat any characters from the
                int     16h             ; keyboard which come along, and
                je      NoKbd           ; handle ctrl-C as appropriate.
                getc

NoKbd:          mov     ah, 84h         ;Now wait for any button to be
                mov     dx, 0           ; pressed.
                int     15h
                and     al, 0F0h
                cmp     al, 0F0h
                je      Wait4Press

                ret
Wait4Button     endp
Installation    ends

sseg            segment para stack 'STACK'
                word    256 dup (0)
endstk          word    ?
sseg            ends

zzzzzzseg       segment para public 'zzzzzzseg'
Heap            byte    1024 dup (0)
zzzzzzseg       ends
                end     Main
```

24.8 Summary

The PC's game adapter card lets you connect a wide variety of game related input devices to your PC. Such devices include digital joysticks, paddles, analog joysticks, steering wheels, yokes, and more. Paddle input devices provide one degree of freedom, joysticks provide two degrees of freedom along an (X,Y) axis pair. Steering wheels and yokes also provide two degrees of freedom, though they are designed for different types of games. For more information on these input devices, see

- "Typical Game Devices" on page 1255

Most game input devices connect to the PC through the game adapter card. This device provides for up to four digital (switch) inputs and four analog (resistive) inputs. This device appears as a single I/O location in the PC's I/O address space. Four of the bits at this port correspond to the four switches, four of the inputs provide the status of the timer pulses from the 558 chip for the analog inputs. The switches you

can read directly from the port; to read the analog inputs, you must create a timing loop to count how long it takes for the pulse associated with a particular device to go from high to low. For more information on the game adapter hardware, see:

- "The Game Adapter Hardware" on page 1257

Programming the game adapter would be a simple task except that you will get different readings for the same relative pot position with different game adapter cards, game input devices, computer systems, and software. The real trick to programming the game adapter is to produce consistent results, regardless of the actual hardware in use. If you can live with raw input values, the BIOS provides two functions to read the switches and the analog inputs. However, if you need normalized values, you will probably have to write your own code. Still, writing such code is very easy if you remember some basic high school algebra. So see how this is done, check out

- "Using BIOS' Game I/O Functions" on page 1259
- "Writing Your Own Game I/O Routines" on page 1260

As with the other devices on the PC, there is a problem with accessing the game adapter hardware directly, such code will not work with game input hardware that doesn't adhere strictly to the original PC's design criteria. Fancy game input devices like the Thrustmaster joystick and the CH Product's FlightStick Pro will require you to write special software drivers. Furthermore, your basic joystick code may not even work with future devices, even if they provide a minimal set of features compatible with standard game input devices. Unfortunately, the BIOS services are very slow and not very good, so few programmers make BIOS calls, allowing third party developers to provide replacement device drivers for their game devices. To help alleviate this problem, this chapter presents the Standard Game Device Input application programmer's interface – a set of functions specifically designed to provide an extensible, portable, system for game input device programmers. The current specification provides for up to 256 digital and 256 analog input devices and is easily extended to handle output devices and other input devices as well. For the details, see

- "The Standard Game Device Interface (SGDI)" on page 1262
- "Application Programmer's Interface (API)" on page 1262

Since this chapter introduces the SGDI driver, there aren't many SGDI drivers provided by game adapter manufacturers at this point. So if you write software that makes SGDI driver calls, you will find that there are few machines that will have an SGDI TSR in memory. Therefore, this chapter provides SGDI drivers for the standard game adapter card and the standard input devices. It also provides an SGDI driver for the CH Products' FlightStick Pro joystick. To obtain these freely distributable drivers, see

- "An SGDI Driver for the Standard Game Adapter Card" on page 1265
- "An SGDI Driver for the CH Products' Flight Stick Pro™" on page 1280

This chapter concludes with an example of a semiresident program that makes SGDI calls. This program, that patches the popular XWing game, provides full support for the CH Product's FlightStick Pro in XWing. This program demonstrates many of the features of an SGDI driver as well as providing and example of how to patch a commercially available game. For the explanation and the source code, see

- "Patching Existing Games" on page 1293

Optimizing Your Programs Chapter 25

Since program optimization is generally one of the last steps in software development, it is only fitting to discuss program optimization in the last chapter of this text. Scanning through other texts that cover this subject, you will find a wide variety of opinions on this subject. Some texts and articles ignore instruction sets altogether and concentrate on finding a better algorithm. Other documents assume you've already found the best algorithm and discuss ways to select the "best" sequence of instructions to accomplish the job. Others consider the CPU architecture and describe how to "count cycles" and pair instructions (especially on superscalar processors or processes with pipelines) to produce faster running code. Others, still, consider the system architecture, not just the CPU architecture, when attempting to decide how to optimize your program. Some authors spend a lot of time explaining that their method is the "one true way" to faster programs. Others still get off on a software engineering tangent and start talking about how time spent optmizing a program isn't worthwhile for a variety of reasons. Well, this chapter is not going to present the "one true way," nor is it going to spend a lot of time bickering about certain optimization techniques. It will simply present you with some examples, options, and suggestions. Since you're on your own after this chapter, it's time for you to start making some of your own decisions. Hopefully, this chapter can provide suitable information so you can make correct decisions.

25.0 Chapter Overview

25.1 When to Optimize, When Not to Optimize

The optimization process is not cheap. If you develop a program and then determine that it is too slow, you may have to redesign and rewrite major portions of that program to get acceptable performance. Based on this point alone, the world often divides itself into two camps – those who optimize early and those who optimize late. Both groups have good arguements; both groups have some bad arguements. Let's take a look at both sides of this arguement.

The "optimize late" (OL) crowd uses the 90/10 arguement: 90% of a program's execution time is spent in 10% of the code[1]. If you try to optimize every piece of code you write (that is, optimize the code before you know that it needs to be optimized), 90% of your effort will go to waste. On the other hand, if you write the code in a normal fashion first and then go in an optimize, you can improve your program's performance with less work. After all, if you *completely removed* the 90% portion of your program, your code would only run about 10% faster. On the other hand, if you completely remove that 10% portion, your program will run about 10 times faster. The math is obviously in favor of attacking the 10%. The OL crowd claims that you should write your code with only the normal attention to performance (i.e., given a choice between an $O(n^2)$ and an O(n lg n) algorithm, you should choose the latter). Once the program is working correctly you can go back and concentrate your efforts on that 10% of the code that takes all the time.

The OL arguements are persuasive. Optimization is a laborious and difficult process. More often that not there is no clear-cut way to speed up a section of code. The only way to determine which of several different options is better is to actually code them all up and compare them. Attempting to do this on the entire program is impractical. However, if you can find that 10% of the code and optimize that, you've reduced your workload by 90%, very inviting indeed. Another good arguement the OL group uses is that few programmers are capable of anticipating where the time will be spent in a program. Therefore, the only real way to determine where a program spends its time is to *instrument it* and measure which functions consume the most time. Obviously, you must have a working program before you can do this. Once

1. Some people prefer to call this the 80/20 rule: 80% of the time is spent in 20% of the code, to be safer in their esitmates. The exact numbers don't matter. What is important is that most of a program's execution time is spent in a small amount of the code.

again, they argue that any time spent optimizing the code beforehand is bound to be wasted since you will probably wind up optimizing that 90% that doesn't need it.

There are, however, some very good counter arguments to the above. First, when most OL types start talking about the 90/10 rule, there is this implicit suggestion that this 10% of the code appears as one big chunk in the middle of the program. A good programmer, like a good surgeon, can locate this malignant mass, cut it out, and replace with with something much faster, thus boosting the speed of your program with only a little effort. Unfortunately, this is not often the case in the real world. In real programs, that 10% of the code that takes up 90% of the execution time is often spread all over your program. You'll get 1% here, 0.5% over there, a "gigantic" 2.5% in one function, and so on. Worse still, optimizing 1% of the code within one function often requires that you modify some of the other code as well. For example, rewriting a function (the 1%) to speed it up quite a bit may require changing the way you pass parameters to that function. This may require rewriting several sections of code outside that slow 10%. So often you wind up rewriting much more than 10% of the code in order to speed up that 10% that takes 90% of the time.

Another problem with the 90/10 rule is that it works on percentages, and the percentages change during optimization. For example, suppose you located a single function that was consuming 90% of the execution time. Let's suppose you're Mr. Super Programmer and you managed to speed this routine up by a factor of two. Your program will now take about 55% of the time to run before it was optimized[2]. If you triple the speed of this routine, your program takes a total of 40% of the original time to execution. If you are really great and you manage to get that function running nine times faster, your program now runs in 20% of the original time, i.e., five times faster.

Suppose you could get that function running nine times faster. Notice that the 90/10 rule no longer applies to your program. 50% of the execution time is spent in 10% of your code, 50% is spent in the other 90% of your code. And if you've managed to speed up that one function by 900%, it is very unlikely you're going to squeeze much more out of it (unless it was *really* bad to begin with). Is it worthwhile messing around with that other 90% of your code? You bet it is. After all, you can improve the performance of your program by 25% if you double the speed of that other code. Note, however, that you only get a 25% performance boost *after* you optimized the 10% as best you could. Had you optimized the 90% of your program first, you would only have gotten a 5% performance improvement; hardly something you'd write home about. Nonetheless, you can see some situations where the 90/10 rule obviously doesn't apply and you can see some cases where optimizing that 90% can produce a good boost in performance. The OL group will smile and say "see, that's the benefit of optimizing late, you can optimize in stages and get just the right amount of optimization you need."

The optimize early (OE) group uses the flaw in percentage arithmetic to point out that you will probably wind up optimizing a large portion of your program anyway. So why not work all this into your design in the first place? A big problem with the OL strategy is that you often wind up designing and writing the program twice – once just to get it functional, the second time to make it practical. After all, if you're going to have to rewrite that 90% anyway, why not write it fast in the first place? The OE people also point out that although programmers are notoriously bad at determining where a program spends most of its time, there are some obvious places where they know there will be performance problems. Why wait to discover the obvious? Why not handle such problem areas early on so there is less time spent measuring and optimizing that code?

Like so many other arguements in Software Engineering, the two camps become quite polarized and swear by a totally pure approach in either direction (either all OE or all OL). Like so many other arguements in Computer Science, the truth actually lies somewhere between these two extremes. Any project where the programmer set out to design the perfect program without worry about performance until the end is doomed. Most programmers in this scenario write *terribly slow* code. Why? Because it's easier to do so and they can always "solve the performance problem during the optimization phase." As a result, the 90% portion of the program is often so slow that even if the time of the other 10% were reduced to zero,

2. Figure the 90% of the code originally took one unit of time to execute and the 10% of the code originally took nine units of time to exeute. If we cut the execution time of the of the 10% in half, we now have 1 unit plus 4.5 units = 5.5 units out of 10 or 55%.

the program would still be way too slow. On the other hand, the OE crowd gets so caught up in writing the best possible code that they miss deadlines and the product may never ship.

There is one undeniable fact that favors the OL arguement – optimized code is difficult to understand and maintain. Furthermore, it often contains bugs that are not present in the unoptimized code. Since incorrect code is unacceptable, even if it does run faster, one very good arguement against optimizing early is the fact that testing, debugging, and quality assurance represent a large portion of the program development cycle. Optimizing early may create so many additional program errors that you lose any time saved by not having to optimize the program later in the development cycle.

The correct time to optimize a program is, well, at the correct time. Unfortunately, the "correct time" varies with the program. However, the first step is to develop program performance requirements along with the other program specifications. The system analyst should develop target response times for all user interactions and computations. During development and testing, programmers have a target to shoot for, so they can't get lazy and wait for the optimization phase before writing code that performs reasonably well. On the other hand, they also have a target to shoot for and once the code is running fast enough, they don't have to waste time, or make their code less maintainable; they can go on and work on the rest of the program. Of course, the system analyst could misjudge performance requirements, but this won't happen often with a good system design.

Another consideration is when to perform *what*. There are several types of optimizations you can perform. For example, you can rearrange instructions to avoid hazards to double the speed of a piece of code. Or you could choose a different algorithm that could run twice as fast. One big problem with optimization is that it is not a single process and many types of optimizations are best done later rather than earlier, or vice versa. For example, choosing a good algorithm is something you should do early on. If you decide to use a better algorithm *after* implementing a poor one, most of the work on the code implementing the old algorithm is lost. Likewise, instruction scheduling is one of the last optimizations you should do. Any changes to the code after rearranging instructions for performance may force you to spend time rearranging them again later. Clearly, the lower level the optimization (i.e., relying upon CPU or system parameters), the later the optimization should be. Conversely, the higher level the optimization (e.g., choice of algorithm), the sooner should be the optimization. In all cases, though, you should have target performance values in mind while developing code.

25.2 How Do You Find the Slow Code in Your Programs?

Although there are problems with the 90/10 rule, the concept behind it is basically solid – programs tend to spend a large amount of their time executing only a small percentage of the code. Clearly, you should optimize the slowest portion of your code first. The only problem is how does one find the slowest code in a program?

There are four common techniques programmers use to find the "hot spots" (the places where programs spend most of their time). The first is by trial and error. The second is to optimize everything. The third is to analyze the program. The fourth is to use a *profiler* or other software monitoring tool to measure the performance of various parts of a program. After locating a hot spot, the programmer can attempt to analyze that section of the program.

The trial and error technique is, unfortunately, the most common strategy. A programmer will speed up various parts of the program by making educated guesses about where it is spending most of its time. If the programmer guesses right, the program will run much faster after optimization. Experienced programmers often use this technique successfully to quickly locate and optimize a program. When the programmer guesses correctly, this technique minimizes the amount of time spent looking for hot spots in a program. Unfortunately, most programmers make fairly poor guesses and wind up optimizing the wrong sections of code. Such effort often goes to waste since optimizing the *wrong* 10% will not improve performance significantly. One of the prime reasons this technique fails so often is that it is often the first choice of inexperienced programmers who cannot easily recognize slow code. Unfotunately, they are probably

unaware of other techniques, so rather than try a structured approach, they start making (often) uneducated guesses.

Another way to locate and optimize the slow portion of a program is to optimize everything. Obviously, this technique does not work well for large programs, but for short sections of code it works reasonably well. Later, this text will provide a short example of an optimization problem and will use this technique to optimize the program. Of course, for large programs or routines this may not be a cost effective approach. However, where appropriate it can save you time while optimizing your program (or at least a portion of your program) since you will not need to carefully analyze and measure the performance of your code. By optimizing everything, you are sure to optimize the slow code.

The analysis method is the most difficult of the four. With this method, you study your code and determine where it will spend most of its time based on the data you expect it to process. In theory, this is the best technique. In practice, human beings generally demonstrate a distaste for such analysis work. As such, the analysis is often incorrect or takes too long to complete. Furthermore, few programmers have much experience studying their code to determine where it is spending most of its time, so they are often quite poor at locating hot spots by studying their listings when the need arises.

Despite the problems with program analysis, this is the first technique you should always use when attempting to optimize a program. Almost all programs spend most of their time executing the body of a loop or recursive function calls. Therefore, you should try to locate all recursive function calls and loop bodies (especially nested loops) in your program. Chances are very good that a program will be spending most of its time in one of these two areas of your program. Such spots are the first to consider when optimizing your programs.

Although the analytical method provides a good way to locate the slow code in a program, analyzing program is a slow, tedious, and boring process. It is very easy to completely miss the most time consuming portion of a program, especially in the presence of indirectly recursive function calls. Even locating time consuming nested loops is often difficult. For example, you might not realize, when looking at a loop within a procedure, that it is a nested loop by virtue of the fact that the calling code executes a loop when calling the procedure. In theory, the analytical method should always work. In practice, it is only marginally successful given that fallible humans are doing the analysis. Nevertheless, some hot spots are easy to find through program analysis, so your first step when optimizing a program should be analysis.

Since programmers are notoriously bad at analyzing programs to find their hot spots, it would make since to try an automate this process. This is precisely what a *profiler* can do for you. A profiler is a small program that measures how long your code spends in any one portion of the program. A profiler typically works by interrupting your code periodically and noting the return address. The profiler builds a histogram of interrupt return addresses (generally rounded to some user specified value). By studying this histogram, you can determine where the program spends most of its time. This tells you which sections of the code you need to optimize. Of course, to use this technique, you will need a profiler program. Borland, Microsoft, and several other vendors provide profilers and other optimization tools.

25.3 Is Optimization Necessary?

Except for fun and education, you should never approach a project with the attitude that you are going to get maximal performance out of your code. Years ago, this was an important attitude because that's what it took to get anything decent running on the slow machines of that era. Reducing the run time of a program from ten minutes to ten seconds made many programs commercially viable. On the other hand, speeding up a program that takes 0.1 seconds to the point where it runs in a millisecond is often pointless. You will waste a lot of effort improving the performance, yet few people will notice the difference.

This is not to say that speeding up programs from 0.1 seconds to 0.001 seconds is never worthwhile. If you are writing a data capture program that requires you to take a reading every millisecond, and it can only handle ten readings per second as currently written, you've got your work cut out for you. Further-

more, even if your program runs fast enough already, there are reasons why you would want to make it run twice as fast. For example, suppose someone can use your program in a multitasking environment. If you modify your program to run twice as fast, the user will be able to run another program along side yours and not notice the performance degradation.

However, the thing to always keep in mind is that you need to write software that is *fast enough*. Once a program produces results instantaneously (or so close to instantaneous that the user can't tell), there is little need to make it run any faster. Since optimization is an expensive and error prone process, you want to avoid it as much as possible. Writing programs that run faster than fast enough is a waste of time. However, as is obvious from the set of bloated application programs you'll find today, this really isn't a problem, most programming produce code that is way too slow, not way too fast.

A common reason stated for not producing optimal code is advancing hardware design. Many programmers and managers feel that the high-end machines they develop software on today will be the mid-range machines two years from now when they finally release their software. So if they design their software to run on today's very high-end machines, it will perform okay on midrange machines when they release their software.

There are two problems with the approach above. First, the operating system running on those machines two years from now will gobble a large part of the machine's resources (including CPU cycles). It is interesting to note that today's machines are hundreds of times faster than the original 8088 based PCs, yet many applications actually run *slower* than those that ran on the original PC. True, today's software provides many more features beyond what the original PC provided, but that's the whole point of this arguement – customers will demand features like multiple windows, GUI, pull-down menus, etc., that all consume CPU cycles. You cannot assume that newer machines will provide extra clock cycles so your slow code will run faster. The OS or user interface to your program will wind up eating those extra available clock cycles.

So the first step is to realistically determine the performance requirements of your software. Then write your software to meet that performance goal. If you fail to meet the performance requirements, then it is time to optimize your program. However, you shouldn't waste additional time optimizing your code once your program meets or exceed the performance specifications.

25.4 The Three Types of Optimization

There are three forms of optimization you can use when improving the performance of a program. They are choosing a better algorithm (high level optimization), implementing the algorithm better (a medium level optmization), and "counting cycles" (a low level optimization). Each technique has its place and, generally, you apply them at different points in the development process.

Choosing a better algorithm is the most highly touted optimization technique. Alas it is the technique used least often. It is easy for someone to announce that you should always find a better algorithm if you need more speed; but finding that algorithm is a little more difficult. First, let us define an algorithm change as using a fundamentally different technique to solve the problem. For example, switching from a "bubble sort" algorithm to a "quick sort" algorithm is a good example of an algorithm change. Generally, though certainly not always, changing algorithms means you use a program with a better Big-Oh function[3] For example, when switching from the bubble sort to the quick sort, you are swapping an algorithm with an $O(n^2)$ running time for one with an O(n lg n) expected running time.

You must remember the restrictions on Big-Oh functions when comparing algorithms. The value for *n* must be sufficiently large to mask the effect of hidden constant. Furthermore, Big-Oh analysis is usually *worst-case* and may not apply to your program. For example, if you wish to sort an array that is "nearly" sorted to begin with, the bubble sort algorithm is usually much faster than the quicksort algorithm, regard-

3. Big-Oh function are approximations of the running time of a program.

less of the value for *n*. For data that is almost sorted, the bubble sort runs in almost O(n) time whereas the quicksort algorithm runs in $O(n^2)$ time[4].

The second thing to keep in mind is the constant itself. If two algorithms have the same Big-Oh function, you cannot determine any difference between the two based on the Big-Oh analysis. This does not mean that they will take the same amount of time to run. Don't forget, in Big-Oh analysis we throw out all the low order terms and multiplicative constants. The asymptotic notation is of little help in this case.

To get truly phenomenal performance improvements requires an algorithmic change to your program. However, discovering an O(n lg n) algorithm to replace your $O(n^2)$ algorithm is often difficult if a published solution does not already exist. Presumably, a well-designed program is not going to contain many obvious algorithms you can dramatically improve (if they did, they wouldn't be well-designed, now, would they?). Therefore, attempting to find a better algorithm may not prove successful. Nevertheless, it is always the first step you should take because the following steps operate on the algorithm you have. If you perform the other steps on a bad algorithm and then discover a better algorithm later, you will have to repeat these time-consumings steps all over again on the new algorithm.

There are two steps to discovering a new algorithms: research and development. The first step is to see if you can find a better solution in the existing literature. Failing that, the second step is to see if you can develop a better algorithm on your own. The key thing is to budget an appropriate amount of time to these two activities. Research is an open-ended process. You can always read one more book or article. So you've got to decide how much time you're going to spend looking for an existing solution. This might be a few hours, days, weeks, or months. Whatever you feel is cost-effective. You then head to the library (or your bookshelf) and begin looking for a better solution. Once your time expires, it is time to abandon the research approach unless you are sure you are on the right track in the material you are studying. If so, budget a little more time and see how it goes. At some point, though, you've got to decide that you probably won't be able to find a better solution and it is time to try to develop a new one on your own.

While searching for a better solution, you should study the papers, texts, articles, etc., exactly as though you were studying for an important test. While it's true that much of what you study will not apply to the problem at hand, you are learning things that will be useful in future projects. Furthermore, while someone may not provide the solution you need, they may have done some work that is headed in the same direction that you are and could provide some good ideas, if not the basis, for your own solution. However, you must always remember that the job of an engineer is to provide a cost-effective solution to a problem. If you waste too much time searching for a solution that may not appear anywhere in the literature, you will cause a cost overrun on your project. So know when it's time to "hang it up" and get on with the rest of the project.

Developing a new algorithm on your own is also open-ended. You could literally spend the rest of your life trying to find an efficient solution to an intractible problem. So once again, you need to budget some time for this process accordingly. Spend the time wisely trying to develop a better solution to your problem, but once the time is exhausted, it's time to try a different approach rather than waste any more time chasing a "holy grail."

Be sure to use all resources at your disposal when trying to find a better algorithm. A local university's library can be a big help. Also, you should network yourself. Attend local computer club meetings, discuss your problems with other engineers, or talk to interested friends, maybe they're read about a solution that you've missed. If you have access to the Internet, BIX, Compuserve, or other technically oriented on-line services or computerized bulletin board systems, by all means post a message asking for help. With literally millions of users out there, if a better solution exists for your problem, someone has probabaly solved it for you already. A few posts may turn up a solution you were unable to find or develop yourself.

At some point or another, you may have to admit failure. Actually, you may have to admit success – you've already found as good an algorithm as you can. If this is still too slow for your requirements, it may be time to try some other technique to improve the speed of your program. The next step is to see if you

4. Yes, $O(n^2)$. The O(n lg n) rating commonly given the quicksort algorithm is actually the *expected* (average case) analysis, not the worst case analysis.

can provide a better implementation for the algorithm you are using. This optimization step, although independent of language, is where most assembly language programmers produce dramatic performance improvements in their code. A better implementation generally involves steps like unrolling loops, using table lookups rather than computations, eliminating computations from a loop whose value does not change within a loop, taking advantage of machine idioms (such as using a shift or shift and add rather than a multiplication), trying to keep variables in registers as long as possible, and so on. It is surprising how much faster a program can run by using simple techniques like those whose descriptions appear thoughout this text.

As a last resort, you can resort to *cycle counting*. At this level you are trying to ensure that an instruction sequence uses as few clock cycles as possible. This is a difficult optimization to perform because you have to be aware of how many clock cycles each instruction consumes, and that depends on the instruction, the addressing mode in use, the instructions around the current instruction (i.e., pipelining and superscalar effects), the speed of the memory system (wait states and cache), and so on. Needless to say, such optimizations are very tedious and require a very careful analysis of the program and the system on which it will run.

The OL crowd always claims you should put off optimization as long as possible. These people are generally talking about this last form of optimization. The reason is simple: any changes you make to your program after such optimizations may change the interaction of the instructions and, therefore, their execution time. If you spend considerable time scheduling a sequence of 50 instructions and then discover you will need to rewrite that code for one reason or another, all the time you spent carefully scheduling those instructions to avoid hazards is lost. On the other hand, if you wait until the last possible moment to make such optimizations to you code, you will only optimize that code once.

Many HLL programmers will tell you that a good compiler can beat a human being at scheduling instructions and optimizing code. This isn't true. A good compiler will beat a mediocre assembly language program a good part of the time. However, a good compiler won't stand a chance against a good assembly language programmer. After all, the worst that could happen is that the good assembly language programmer will look at the output of the compiler and improve on that.

"Counting cycles" can improve the performance of your programs. On the average, you can speed up your programs by a factor of 50% to 200% by making simple changes (like rearranging instructions). That's the difference between an 80486 and a Pentium! So you shouldn't ignore the possibility of using such optimizations in your programs. Just keep in mind, you should do such optimizations last so you don't wind up redoing them as your code changes.

The rest of this chapter will concentrate on the techniques for improving the implementation of an algorithm, rather than designing a better algorithm or using cycle counting techniques. Designing better algorithms is beyond the scope of this manual (see a good text on algorithm design). Cycle counting is one of those processes that differs from processor to processor. That is, the optimization techniques that work well for the 80386 fail on a 486 or Pentium chip, and vice versa. Since Intel is constantly producing new chips, requring different optimization techniques, listing those techniques here would only make that much more material in this book outdated. Intel publishes such optimization hints in their processor programmer reference manuals. Articles on optimizing assembly language programs often appear in technical magazines like Dr. Dobb's Journal, you should read such articles and learn all the current optimization techniques.

25.5 Improving the Implementation of an Algorithm

One easy way to partially demonstrate how to optimize a piece of code is to provide an example of some program and the optimization steps you can apply to that program. This section will present a short program that *blurs* an eight-bit gray scale image. Then, this section will lead though through several optimization steps and show you how to get that program running over 16 times faster.

The following code assumes that you provide it with a file containing a 251x256 gray scale photographic image. The data structure for this file is as follows:

```
Image: array [0..250, 0..255] of byte;
```

Each byte contains a value in the range 0..255 with zero denoting black, 255 representing white, and the other values representing even shades of gray between these two extremes.

The blurring algorithm averages a pixel[5] with its eight closest neighbors. A single blur operation applies this average to all interior pixels of an image (that is, it does not apply to the pixels on the boundary of the image because they do not have the same number of neighbors as the other pixels). The following Pascal program implements the blurring algorithm and lets the user specify the amount of blurring (by looping through the algorithm the number of times the user specifies)[6]:

```
program PhotoFilter(input,output);

(* Here is the raw file data type produced by the Photoshop program *)

type
 image = array [0..250] of array [0..255] of byte;

(* The variables we will use. Note that the “datain” and “dataout” *)
(* variables are pointers because Turbo Pascal will not allow us to *)
(* allocate more than 64K data in the one global data segment it *)
(* supports. *)

var
 h,i,j,k,l,sum,iterations:integer;
 datain, dataout: ^image;
 f,g:file of image;

begin

 (* Open the files and real the input data *)

 assign(f, ‘roller1.raw’);
 assign(g, ‘roller2.raw’);
 reset(f);
 rewrite(g);
 new(datain);
 new(dataout);
 read(f,datain^);

 (* Get the number of iterations from the user *)

 write(‘Enter number of iterations:’);
 readln(iterations);

 writeln(‘Computing result’);

 (* Copy the data from the input array to the output array. *)
 (* This is a really lame way to copy the border from the *)
 (* input array to the output array. *)

 for i := 0 to 250 do
        for j := 0 to 255 do
              dataout^ [i][j] := datain^ [i][j];

 (* Okay, here’s where all the work takes place. The outside     *)
 (* loop repeats this blurring operation the number of           *)
 (* iterations specified by the user.                            *)

 for h := 1 to iterations do begin

        (* For each row except the first and the last, compute  *)
        (* a new value for each element.                        *)

        for i := 1 to 249 do
```

5. Pixel stands for “picture element.” A pixel is an element of the Image array defined above.
6. A comparable C program appears on the diskette accompanying the lab manual.

```
                    (* For each column except the first and the last, com-    *)
                    (* pute a new value for each element.                     *)

                    for j := 1 to 254 do begin

                              (* For each element in the array, compute a new
                                 blurred value by adding up the eight cells
                                 around an array element along with eight times
                                 the current cell's value. Then divide this by
                                 sixteen to compute a weighted average of the
                                 nine cells forming a square around the current
                                 cell. The current cell has a 50% weighting,
                                 the other eight cells around the current cel
                                 provide the other 50% weighting (6.25% each). *)

                              sum := 0;
                               for k := -1 to 1 do
                                  for l := -1 to 1 do
                                            sum := sum + datain^ [i+k][j+l];

                              (* Sum currently contains the sum of the nine     *)
                              (* cells, add in seven times the current cell so  *)
                              (* we get a total of eight times the current cell. *)

                              dataout^ [i][j] := (sum + datain^ [i][j]*7) div 16;

                    end;

                    (* Copy the output cell values back to the input cells    *)
                    (* so we can perform the blurring on this new data on     *)
                    (* the next iteration.                                    *)

                    for i := 0 to 250 do
                     for j := 0 to 255 do
                              datain^ [i][j] := dataout^ [i][j];

 end;

 writeln('Writing result');
 write(g,dataout^);
 close(f);
 close(g);

end.
```

The Pascal program above, compiled with Turbo Pascal v7.0, takes 45 seconds to compute 100 iterations of the blurring algorithm. A comparable program written in C and compiled with Borland C++ v4.02 takes 29 seconds to run. The same source file compiled with Microsoft C++ v8.00 runs in 21 seconds. Obviously the C compilers produce better code than Turbo Pascal. It took about three hours to get the Pascal version running and tested. The C versions took about another hour to code and test. The following two images provide a "before" and "after" example of this program's function:

Before blurring:

After blurring (10 iterations):

The following is a crude translation from Pascal directly into assembly language of the above program. It requires 36 seconds to run. Yes, the C compilers did a better job, but once you see how bad this code is, you'll wonder what it is that Turbo Pascal is doing to run so slow. It took about an hour to translate the Pascal version into this assembly code and debug it to the point it produced the same output as the Pascal version.

```
; IMGPRCS.ASM
;
; An image processing program.
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
```

```
;
;       Performance comparisons (66 MHz 80486 DX/2 system).
;
;       This code-                      36 seconds.
;       Borland Pascal v7.0-            45 seconds.
;       Borland C++ v4.02-              29 seconds.
;       Microsoft C++ v8.00-            21 seconds.

              .xlist
              include    stdlib.a
              includelib stdlib.lib
              .list
              .286

dseg          segment    para public 'data'

; Loop control variables and other variables:

h             word       ?
i             word       ?
j             word       ?
k             word       ?
l             word       ?
sum           word       ?
iterations    word       ?

; File names:

InName        byte       "roller1.raw",0
OutName       byte       "roller2.raw",0

dseg          ends


; Here is the input data that we operate on.

InSeg         segment    para public 'indata'

DataIn        byte       251 dup (256 dup (?))

InSeg         ends


; Here is the output array that holds the result.

OutSeg        segment    para public 'outdata'

DataOut       byte       251 dup (256 dup (?))

OutSeg        ends



cseg          segment    para public 'code'
              assume     cs:cseg, ds:dseg

Main          proc
              mov        ax, dseg
              mov        ds, ax
              meminit

              mov        ax, 3d00h         ;Open input file for reading.
              lea        dx, InName
              int        21h
              jnc        GoodOpen
              print
              byte       "Could not open input file.",cr,lf,0
              jmp        Quit

GoodOpen:     mov        bx, ax            ;File handle.
              mov        dx, InSeg         ;Where to put the data.
              mov        ds, dx
              lea        dx, DataIn
```

```
                mov     cx, 256*251     ;Size of data file to read.
                mov     ah, 3Fh
                int     21h
                cmp     ax, 256*251     ;See if we read the data.
                je      GoodRead
                print
                byte    "Did not read the file properly",cr,lf,0
                jmp     Quit

GoodRead:       mov     ax, dseg
                mov     ds, ax
                print
                byte    "Enter number of iterations: ",0
                getsm
                atoi
                free
                mov     iterations, ax
                print
                byte    "Computing Result",cr,lf,0

; Copy the input data to the output buffer.

                mov     i, 0
iloop0:         cmp     i, 250
                ja      iDone0
                mov     j, 0
jloop0:         cmp     j, 255
                ja      jDone0

                mov     bx, i           ;Compute index into both
                shl     bx, 8           ; arrays using the formula
                add     bx, j           ; i*256+j (row major).

                mov     cx, InSeg       ;Point at input segment.
                mov     es, cx
                mov     al, es:DataIn[bx] ;Get DataIn[i][j].

                mov     cx, OutSeg      ;Point at output segment.
                mov     es, cx
                mov     es:DataOut[bx], al ;Store into DataOut[i][j]

                inc     j               ;Next iteration of j loop.
                jmp     jloop0

jDone0:         inc     i               ;Next iteration of i loop.
                jmp     iloop0

iDone0:

; for h := 1 to iterations-

                mov     h, 1
hloop:          mov     ax, h
                cmp     ax, iterations
                ja      hloopDone



; for i := 1 to 249 -

                mov     i, 1
iloop:          cmp     i, 249
                ja      iloopDone

; for j := 1 to 254 -
                mov     j, 1
jloop:          cmp     j, 254
                ja      jloopDone


; sum := 0;
; for k := -1 to 1 do for l := -1 to 1 do

                mov     ax, InSeg       ;Gain access to InSeg.
```

```
                mov     es, ax

                mov     sum, 0
                mov     k, -1
kloop:          cmp     k, 1
                jg      kloopDone

                mov     l, -1
lloop:          cmp     l, 1
                jg      lloopDone

; sum := sum + datain [i+k][j+l]

                mov     bx, i
                add     bx, k
                shl     bx, 8           ;Multiply by 256.
                add     bx, j
                add     bx, l

                mov     al, es:DataIn[bx]
                mov     ah, 0
                add     Sum, ax

                inc     l
                jmp     lloop

lloopDone:      inc     k
                jmp     kloop


; dataout [i][j] := (sum + datain[i][j]*7) div 16;

kloopDone:      mov     bx, i
                shl     bx, 8           ;*256
                add     bx, j
                mov     al, es:DataIn[bx]
                mov     ah, 0
                imul    ax, 7
                add     ax, sum
                shr     ax, 4           ;div 16

                mov     bx, OutSeg
                mov     es, bx

                mov     bx, i
                shl     bx, 8
                add     bx, j
                mov     es:DataOut[bx], al

                inc     j
                jmp     jloop

jloopDone:      inc     i
                jmp     iloop

iloopDone:
; Copy the output data to the input buffer.

                mov     i, 0
iloop1:         cmp     i, 250
                ja      iDone1
                mov     j, 0
jloop1:         cmp     j, 255
                ja      jDone1

                mov     bx, i           ;Compute index into both
                shl     bx, 8           ; arrays using the formula
                add     bx, j           ; i*256+j (row major).

                mov     cx, OutSeg      ;Point at input segment.
                mov     es, cx
                mov     al, es:DataOut[bx] ;Get DataIn[i][j].

                mov     cx, InSeg       ;Point at output segment.
```

```
                mov         es, cx
                mov         es:DataIn[bx], al ;Store into DataOut[i][j]

                inc         j                 ;Next iteration of j loop.
                jmp         jloop1

jDone1:         inc         i                 ;Next iteration of i loop.
                jmp         iloop1

iDone1:         inc         h
                jmp         hloop

hloopDone:      print
                byte        “Writing result”,cr,lf,0


; Okay, write the data to the output file:

                mov         ah, 3ch           ;Create output file.
                mov         cx, 0             ;Normal file attributes.
                lea         dx, OutName
                int         21h
                jnc         GoodCreate
                print
                byte        “Could not create output file.”,cr,lf,0
                jmp         Quit

GoodCreate:     mov         bx, ax        ;File handle.
                push        bx
                mov         dx, OutSeg        ;Where the data can be found.
                mov         ds, dx
                lea         dx, DataOut
                mov         cx, 256*251       ;Size of data file to write.
                mov         ah, 40h           ;Write operation.
                int         21h
                pop         bx                ;Retrieve handle for close.
                cmp         ax, 256*251       ;See if we wrote the data.
                je          GoodWrite
                print
                byte        “Did not write the file properly”,cr,lf,0
                jmp         Quit

GoodWrite:      mov         ah, 3eh           ;Close operation.
                int         21h


Quit:           ExitPgm                       ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment     para stack ‘stack’
stk             byte        1024 dup (“stack “)
sseg            ends

zzzzzzseg       segment     para public ‘zzzzzz’
LastBytes       byte        16 dup (?)
zzzzzzseg       ends
                end         Main
```

This assembly code is a very straight-forward, line by line translation of the previous Pascal code. Even beginning programmers (who’ve read and understand Chapters Eight and Nine) should easily be able to improve the performance of this code.

While we could run a profiler on this program to determine where the “hot spots” are in this code, a little analysis, particularly of the Pascal version, should make it obvious that there are a lot of nested loops in this code. As Chapter Ten points out, when optimizing code you should always start with the innermost loops. The major change between the code above and the following assembly language version is that we’ve unrolled the innermost loops and we’ve replaced the array index computations with some constant

computations. These minor changes speed up the execution by a factor of six! The assembly version now runs in six seconds rather than 36. A Microsoft C++ version of the same program with comparable optimzations runs in eight seconds. It required nearly four hours to develop, test, and debug this code. It required an additional hour to apply these same modifications to the C version[7].

```
; IMGPRCS2.ASM
;
; An image processing program (First optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
; Version #2: Three major optimizations. (1) used movsd instruction rather
;         than a loop to copy data from DataOut back to DataIn.
;         (2) Used repeat..until forms for all loops. (3) unrolled
;         the innermost two loops (which is responsible for most of
;         the performance improvement).
;
;
;       Performance comparisons (66 MHz 80486 DX/2 system).
;
;       This code-                      6 seconds.
;       Original ASM code-              36 seconds.
;       Borland Pascal v7.0-            45 seconds.
;       Borland C++ v4.02-              29 seconds.
;       Microsoft C++ v8.00-            21 seconds.


;       « Lots of omitted code goes here, see the previous version»


                print
                byte        “Computing Result”,cr,lf,0

; for h := 1 to iterations-

                mov         h, 1
hloop:

; Copy the input data to the output buffer.
; Optimization step #1: Replace with movs instruction.

                push        ds
                mov         ax, OutSeg
                mov         ds, ax
                mov         ax, InSeg
                mov         es, ax
                lea         si, DataOut
                lea         di, DataIn
                mov         cx, (251*256)/4
        rep     movsd
                pop         ds


; Optimization Step #1: Convert loops to repeat..until form.

; for i := 1 to 249 -

                mov         i, 1
iloop:

; for j := 1 to 254 -
```

7. This does not imply that coding this improved algorithm in C was easier. Most of the time on the assembly version was spent trying out several different modifications to see if they actually improved performance. Many modifications did not, so they were removed from the code. The development of the C version benefited from the past work on the assembly version. It was a straight-forward conversion from assembly to C.

```
                mov     j, 1
jloop:


; Optimization. Unroll the innermost two loops:

                mov     bh, byte ptr i ;i is always less than 256.
                mov     bl, byte ptr j ;Computes i*256+j!

                push    ds
                mov     ax, InSeg      ;Gain access to InSeg.
                mov     ds, ax

                mov     cx, 0             ;Compute sum here.
                mov     ah, ch
                mov     cl, ds:DataIn[bx-257];DataIn[i-1][j-1]
                mov     al, ds:DataIn[bx-256];DataIn[i-1][j]
                add     cx, ax
                mov     al, ds:DataIn[bx-255];DataIn[i-1][j+1]
                add     cx, ax
                mov     al, ds:DataIn[bx-1];DataIn[i][j-1]
                add     cx, ax
                mov     al, ds:DataIn[bx+1];DataIn[i][j+1]
                add     cx, ax
                mov     al, ds:DataIn[bx+255];DataIn[i+1][j-1]
                add     cx, ax
                mov     al, ds:DataIn[bx+256];DataIn[i+1][j]
                add     cx, ax
                mov     al, ds:DataIn[bx+257];DataIn[i+1][j+1]
                add     cx, ax

                mov     al, ds:DataIn[bx];DataIn[i][j]
                shl     ax, 3             ;DataIn[i][j]*8
                add     cx, ax
                shr     cx, 4             ;Divide by 16
                mov     ax, OutSeg
                mov     ds, ax
                mov     ds:DataOut[bx], cl
                pop     ds

                inc     j
                cmp     j, 254
                jbe     jloop

                inc     i
                cmp     i, 249
                jbe     iloop

                inc     h
                mov     ax, h
                cmp     ax, Iterations
                jnbe    Done
                jmp     hloop

Done:           print
                byte    “Writing result”,cr,lf,0

;       «More omitted code goes here, see the previous version»
```

The second version above still uses memory variables for most computations. The optimizations applied to the original code were mainly language-independent optimizations. The next step was to begin applying some assembly language specific optimizations to the code. The first optimization we need to do is to move as many variables as possible into the 80x86's register set. The following code provides this optimization. Although this only improves the running time by 2 seconds, that is a 33% improvement (six seconds down to four)!

```
; IMGPRCS.ASM
;
; An image processing program (Second optimization pass).
```

```
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
; Version #2: Three major optimizations. (1) used movsd instruction rather
;        than a loop to copy data from DataOut back to DataIn.
;        (2) Used repeat..until forms for all loops. (3) unrolled
;        the innermost two loops (which is responsible for most of
;        the performance improvement).
; Version #3: Used registers for all variables. Set up segment registers
;        once and for all through the execution of the main loop so
;        the code didn't have to reload ds each time through. Computed
;        index into each row only once (outside the j loop).
;
;
;       Performance comparisons (66 MHz 80486 DX/2 system).
;
;       This code-                      4 seconds.
;       1st optimization pass-          6 seconds.
;       Original ASM code-              36 seconds.


;       «Lots of delete code goes here»

              print
              byte      "Computing Result",cr,lf,0


; Copy the input data to the output buffer.


hloop:        mov       ax, InSeg
              mov       es, ax
              mov       ax, OutSeg
              mov       ds, ax
              lea       si, DataOut
              lea       di, DataIn
              mov       cx, (251*256)/4
        rep   movsd

              assume    ds:InSeg, es:OutSeg
              mov       ax, InSeg
              mov       ds, ax
              mov       ax, OutSeg
              mov       es, ax



              mov       cl, 249
iloop:        mov       bh, cl            ;i*256
              mov       bl, 1             ;Start at j=1.
              mov       ch, 254           ;# of times through loop.
jloop:
              mov       dx, 0             ;Compute sum here.
              mov       ah, dh
              mov       dl, DataIn[bx-257]              ;DataIn[i-1][j-1]
              mov       al, DataIn[bx-256]              ;DataIn[i-1][j]
              add       dx, ax
              mov       al, DataIn[bx-255]              ;DataIn[i-1][j+1]
              add       dx, ax
              mov       al, DataIn[bx-1]                ;DataIn[i][j-1]
              add       dx, ax
              mov       al, DataIn[bx+1]                ;DataIn[i][j+1]
              add       dx, ax
              mov       al, DataIn[bx+255]              ;DataIn[i+1][j-1]
              add       dx, ax
              mov       al, DataIn[bx+256]              ;DataIn[i+1][j]
              add       dx, ax
              mov       al, DataIn[bx+257]              ;DataIn[i+1][j+1]
```

```
                add         dx, ax

                mov         al, DataIn[bx]                  ;DataIn[i][j]
                shl         ax, 3                           ;DataIn[i][j]*8
                add         dx, ax
                shr         dx, 4                           ;Divide by 16
                mov         DataOut[bx], dl

                inc         bx
                dec         ch
                jne         jloop

                dec         cl
                jne         iloop

                dec         bp
                jne         hloop

Done:           print
                byte        "Writing result",cr,lf,0

;       «More deleted code goes here, see the original version»
```

Note that on each iteration, the code above still copies the output data back to the input data. That's almost six and a half megabytes of data movement for 100 iterations! The following version of the blurring program unrolls the hloop twice. The first occurrence copies the data from DataIn to DataOut while computing the blur, the second instance copies the data from DataOut back to DataIn while blurring the image. By using these two code sequences, the program save copying the data from one point to another. This version also maintains some common computations between two adjacent cells to save a few instructions in the innermost loop. This version arranges instructions in the innermost loop to help avoid data hazards on 80486 and later processors. The end result is almost 40% faster than the previous version (down to 2.5 seconds from four seconds).

```
; IMGPRCS.ASM
;
; An image processing program (Third optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
;
; Version #2: Three major optimizations. (1) used movsd instruction rather
;         than a loop to copy data from DataOut back to DataIn.
;         (2) Used repeat..until forms for all loops. (3) unrolled
;         the innermost two loops (which is responsible for most of
;         the performance improvement).
;
; Version #3: Used registers for all variables. Set up segment registers
;         once and for all through the execution of the main loop so
;         the code didn't have to reload ds each time through. Computed
;         index into each row only once (outside the j loop).
;
; Version #4: Eliminated copying data from DataOut to DataIn on each pass.
;         Removed hazards. Maintained common subexpressions. Did some
;         more loop unrolling.
;
;
;       Performance comparisons (66 MHz 80486 DX/2 system, 100 iterations).
;
;       This code-                  2.5 seconds.
;       2nd optimization pass-      4 seconds.
;       1st optimization pass-      6 seconds.
;       Original ASM code-          36 seconds.

;       «Lots of deleted code here, see the original version»
```

```
                print
                byte    "Computing Result",cr,lf,0


                assume  ds:InSeg, es:OutSeg

                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax

; Copy the data once so we get the edges in both arrays.

                mov     cx, (251*256)/4
                lea     si, DataIn
                lea     di, DataOut
        rep     movsd


; "hloop" repeats once for each iteration.

hloop:
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax

; "iloop" processes the rows in the matrices.

                mov     cl, 249
iloop:          mov     bh, cl          ;i*256
                mov     bl, 1           ;Start at j=1.
                mov     ch, 254/2       ;# of times through loop.
                mov     si, bx
                mov     dh, 0           ;Compute sum here.
                mov     bh, 0
                mov     ah, 0

; "jloop" processes the individual elements of the array.
; This loop has been unrolled once to allow the two portions to share
; some common computations.

jloop:

; The sum of DataIn [i-1][j] + DataIn[i-1][j+1] + DataIn[i+1][j] +
; DataIn [i+1][j+1] will be used in the second half of this computation.
; So save its value in a register (di) until we need it again.

                mov     dl, DataIn[si-256]          ;[i-1,j]
                mov     al, DataIn[si-255]          ;[i-1,j+1]
                mov     bl, DataIn[si+257]          ;[i+1,j+1]
                add     dx, ax
                mov     al, DataIn[si+256]          ;[I+1,j]
                add     dx, bx
                mov     bl, DataIn[si+1]            ;[i,j+1]
                add     dx, ax
                mov     al, DataIn[si+255]          ;[i+1,j-1]

                mov     di, dx                      ;Save partial result.

                add     dx, bx
                mov     bl, DataIn[si-1]            ;[i,j-1]
                add     dx, ax
                mov     al, DataIn[si]              ;[i,j]
                add     dx, bx
                mov     bl, DataIn[si-257]          ;[i-1,j-1]
                shl     ax, 3                       ;DataIn[i,j] * 8.
                add     dx, bx
                add     dx, ax
                shr     ax, 3                       ;Restore DataIn[i,j].
                shr     dx, 4                       ;Divide by 16.
                add     di, ax
                mov     DataOut[si], dl
```

```
; Okay, process the next cell over. Note that we've got a partial sum
; sitting in DI already. Don't forget, we haven't bumped SI at this point,
; so the offsets are off by one. (This is the second half of the unrolled
; loop.)

                mov     dx, di                          ;Partial sum.
                mov     bl, DataIn[si-254]              ;[i-1,j+1]
                mov     al, DataIn[si+2]                ;[i,j+1]
                add     dx, bx
                mov     bl, DataIn[si+258]              ;[i+1,j+1];
                add     dx, ax
                mov     al, DataIn[si+1]                ;[i,j]
                add     dx, bx
                shl     ax, 3                           ;DataIn[i][j]*8
                add     si, 2                           ;Bump array index.
                add     dx, ax
                mov     ah, 0                           ;Clear for next iter.
                shr     dx, 4                           ;Divide by 16
                dec     ch
                mov     DataOut[si-1], dl
                jne     jloop

                dec     cl
                jne     iloop

                dec     bp
                je      Done


; Special case so we don't have to move the data between the two arrays.
; This is an unrolled version of the hloop that swaps the input and output
; arrays so we don't have to move data around in memory.

                mov     ax, OutSeg
                mov     ds, ax
                mov     ax, InSeg
                mov     es, ax
                assume  es:InSeg, ds:OutSeg

hloop2:

                mov     cl, 249
iloop2:         mov     bh, cl
                mov     bl, 1
                mov     ch, 254/2
                mov     si, bx
                mov     dh, 0
                mov     bh, 0
                mov     ah, 0
jloop2:
                mov     dl, DataOut[si-256]
                mov     al, DataOut[si-255]
                mov     bl, DataOut[si+257]
                add     dx, ax
                mov     al, DataOut[si+256]
                add     dx, bx
                mov     bl, DataOut[si+1]
                add     dx, ax
                mov     al, DataOut[si+255]

                mov     di, dx

                add     dx, bx
                mov     bl, DataOut[si-1]
                add     dx, ax
                mov     al, DataOut[si]
                add     dx, bx
                mov     bl, DataOut[si-257]
                shl     ax, 3
                add     dx, bx
                add     dx, ax
                shr     ax, 3
                shr     dx, 4
                mov     DataIn[si], dl
```

```
                mov         dx, di
                mov         bl, DataOut[si-254]
                add         dx, ax
                mov         al, DataOut[si+2]
                add         dx, bx
                mov         bl, DataOut[si+258]
                add         dx, ax
                mov         al, DataOut[si+1]
                add         dx, bx
                shl         ax, 3
                add         si, 2
                add         dx, ax
                mov         ah, 0
                shr         dx, 4
                dec         ch
                mov         DataIn[si-1], dl
                jne         jloop2

                dec         cl
                jne         iloop2

                dec         bp
                je          Done2
                jmp         hloop


; Kludge to guarantee that the data always resides in the output segment.

Done2:
                mov         ax, InSeg
                mov         ds, ax
                mov         ax, OutSeg
                mov         es, ax
                mov         cx, (251*256)/4
                lea         si, DataIn
                lea         di, DataOut
        rep     movsd

Done:           print
                byte        "Writing result",cr,lf,0

;       «Lots of deleted code here, see the original program»
```

This code provides a good example of the kind of optimization that scares a lot of people. There is a lot of cycle counting, instruction scheduling, and other crazy stuff that makes program very difficult to read and understand. This is the kind of optimization for which assembly language programmers are famous; the stuff that spawned the phrase "never optimize early." You should never try this type of optimization until you feel you've exhausted all other possibilities. Once you write your code in this fashion, it is going to be very difficult to make further changes to it. By the way, the above code took about 15 hours to develop and debug (debugging took the most time). That works out to a 0.1 second improvement (for 100 iterations) for each hour of work. Although this code certainly isn't optimal yet, it is difficult to justify more time attempting to improve this code by mechanical means (e.g., moving instructions around, etc.) because the performance gains would be so little.

In the four steps above, we've reduced the running time of the assembly code from 36 seconds down to 2.5 seconds. Quite an impressive feat. However, you shouldn't get the idea that this was easy or even that there were only four steps involved. During the actual development of this example, there were many attempts that did not improve performance (in fact, some modifications wound up reducing performance) and others did not improve performance enough to justify their inclusion. Just to demonstrate this last point, the following code included a major change in the way the program organized data. The main loop operates on 16 bit objects in memory rather than eight bit objects. On some machines with large external caches (256K or better) this algorithm provides a slight improvement in performance (2.4 seconds, down from 2.5). However, on other machines it runs slower. Therefore, this code was not chosen as the final implementation:

```
; IMGPRCS.ASM
;
; An image processing program (Fourth optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
;
; Version #2: Three major optimizations. (1) used movsd instruction rather
;        than a loop to copy data from DataOut back to DataIn.
;        (2) Used repeat..until forms for all loops. (3) unrolled
;        the innermost two loops (which is responsible for most of
;        the performance improvement).
;
; Version #3: Used registers for all variables. Set up segment registers
;        once and for all through the execution of the main loop so
;        the code didn't have to reload ds each time through. Computed
;        index into each row only once (outside the j loop).
;
; Version #4: Eliminated copying data from DataOut to DataIn on each pass.
;        Removed hazards. Maintained common subexpressions. Did some
;        more loop unrolling.
;
; Version #5: Converted data arrays to words rather than bytes and operated
;        on 16-bit values. Yielded minimal speedup.
;
;       Performance comparisons (66 MHz 80486 DX/2 system).
;
;       This code-                    2.4 seconds.
;       3rd optimization pass-        2.5 seconds.
;       2nd optimization pass-        4 seconds.
;       1st optimization pass-        6 seconds.
;       Original ASM code-            36 seconds.

             .xlist
             include    stdlib.a
             includelib stdlib.lib
             .list
             .386
             option        segment:use16



dseg         segment    para public 'data'

ImgData      byte       251 dup (256 dup (?))

InName       byte       "roller1.raw",0
OutName      byte       "roller2.raw",0
Iterations   word       0

dseg         ends


; This code makes the naughty assumption that the following
; segments are loaded contiguously in memory! Also, because these
; segments are paragraph aligned, this code assumes that these segments
; will contain a full 65,536 bytes. You cannot declare a segment with
; exactly 65,536 bytes in MASM. However, the paragraph alignment option
; ensures that the extra byte of padding is added to the end of each
; segment.

DataSeg1     segment    para public 'ds1'
Data1a       byte       65535 dup (?)
DataSeg1     ends

DataSeg2     segment    para public 'ds2'
Data1b       byte       65535 dup (?)
DataSeg2     ends
```

```
DataSeg3        segment     para public 'ds3'
Data2a          byte        65535 dup (?)
DataSeg3        ends

DataSeg4        segment     para public 'ds4'
Data2b          byte        65535 dup (?)
DataSeg4        ends



cseg            segment     para public 'code'
                assume      cs:cseg, ds:dseg

Main            proc
                mov         ax, dseg
                mov         ds, ax
                meminit

                mov         ax, 3d00h       ;Open input file for reading.
                lea         dx, InName
                int         21h
                jnc         GoodOpen
                print
                byte        "Could not open input file.",cr,lf,0
                jmp         Quit

GoodOpen:       mov         bx, ax              ;File handle.
                lea         dx, ImgData
                mov         cx, 256*251         ;Size of data file to read.
                mov         ah, 3Fh
                int         21h
                cmp         ax, 256*251         ;See if we read the data.
                je          GoodRead
                print
                byte        "Did not read the file properly",cr,lf,0
                jmp         Quit

GoodRead:       print
                byte        "Enter number of iterations: ",0
                getsm
                atoi
                free
                mov         Iterations, ax
                cmp         ax, 0
                jle         Quit

                printf
                byte        "Computing Result for %d iterations",cr,lf,0
                dword       Iterations


; Copy the data and expand it from eight bits to sixteen bits.
; The first loop handles the first 32,768 bytes, the second loop
; handles the remaining bytes.

                mov         ax, DataSeg1
                mov         es, ax
                mov         ax, DataSeg3
                mov         fs, ax

                mov         ah, 0
                mov         cx, 32768
                lea         si, ImgData
                xor         di, di              ;Output data is at ofs zero.
CopyLoop:       lodsb                           ;Read a byte
                mov         fs:[di], ax         ;Store a word in DataSeg3
                stosw                           ;Store a word in DataSeg1
                dec         cx
                jne         CopyLoop

                mov         di, DataSeg2
```

```
                mov     es, di
                mov     di, DataSeg4
                mov     fs, di
                mov     cx, (251*256) - 32768
                xor     di, di
CopyLoop1:      lodsb                   ;Read a byte
                mov     fs:[di], ax     ;Store a word in DataSeg4
                stosw                   ;Store a word in DataSeg2
                dec     cx
                jne     CopyLoop1

; hloop completes one iteration on the data moving it from Data1a/Data1b
; to Data2a/Data2b

hloop:          mov     ax, DataSeg1
                mov     ds, ax
                mov     ax, DataSeg3
                mov     es, ax

; Process the first 127 rows (65,024 bytes) of the array):

                mov     cl, 127
                lea     si, Data1a+202h ;Start at [1,1]
iloop0:         mov     ch, 254/2       ;# of times through loop.
jloop0:         mov     dx, [si]        ;[i,j]
                mov     bx, [si-200h]   ;[i-1,j]
                mov     ax, dx
                shl     dx, 3           ;[i,j] * 8
                add     bx, [si-1feh]   ;[i-1,j+1]
                mov     bp, [si+2]      ;[i,j+1]
                add     bx, [si+200h]   ;[i+1,j]
                add     dx, bp
                add     bx, [si+202h]   ;[i+1,j+1]
                add     dx, [si-202h]   ;[i-1,j-1]
                mov     di, [si-1fch]   ;[i-1,j+2]
                add     dx, [si-2]      ;[i,j-1]
                add     di, [si+4]      ;[i,j+2]
                add     dx, [si+1feh]   ;[i+1,j-1]
                add     di, [si+204h]   ;[i+1,j+2]
                shl     bp, 3           ;[i,j+1] * 8
                add     dx, bx
                add     bp, ax
                shr     dx, 4           ;Divide by 16.
                add     bp, bx
                mov     es:[si], dx     ;Store [i,j] entry.
                add     bp, di
                add     si, 4           ;Affects next store operation!
                shr     bp, 4           ;Divide by 16.
                dec     ch
                mov     es:[si-2], bp   ;Store [i,j+1] entry.
                jne     jloop0

                add     si, 4           ;Skip to start of next row.

                dec     cl
                jne     iloop0

; Process the last 124 rows of the array). This requires that we switch from
; one segment to the next. Note that the segments overlap.

                mov     ax, DataSeg2
                sub     ax, 40h         ;Back up to last 2 rows in DS2
                mov     ds, ax
                mov     ax, DataSeg4
                sub     ax, 40h         ;Back up to last 2 rows in DS4
                mov     es, ax

                mov     cl, 251-127-1   ;Remaining rows to process.
                mov     si, 202h        ;Continue with next row.
iloop1:         mov     ch, 254/2       ;# of times through loop.
jloop1:         mov     dx, [si]        ;[i,j]
                mov     bx, [si-200h]   ;[i-1,j]
                mov     ax, dx
                shl     dx, 3           ;[i,j] * 8
```

```
                add     bx, [si-1feh]   ;[i-1,j+1]
                mov     bp, [si+2]      ;[i,j+1]
                add     bx, [si+200h]   ;[i+1,j]
                add     dx, bp
                add     bx, [si+202h]   ;[i+1,j+1]
                add     dx, [si-202h]   ;[i-1,j-1]
                mov     di, [si-1fch]   ;[i-1,j+2]
                add     dx, [si-2]      ;[i,j-1]
                add     di, [si+4]      ;[i,j+2]
                add     dx, [si+1feh]   ;[i+1,j-1]
                add     di, [si+204h]   ;[i+1,j+2]
                shl     bp, 3           ;[i,j+1] * 8
                add     dx, bx
                add     bp, ax
                shr     dx, 4           ;Divide by 16
                add     bp, bx
                mov     es:[si], dx     ;Store [i,j] entry.
                add     bp, di
                add     si, 4           ;Affects next store operation!
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp   ;Store [i,j+1] entry.
                jne     jloop1

                add     si, 4           ;Skip to start of next row.

                dec     cl
                jne     iloop1

                mov     ax, dseg
                mov     ds, ax
                assume  ds:dseg

                dec     Iterations
                je      Done0

; Unroll the iterations loop so we can move the data from DataSeg2/4 back
; to DataSeg1/3 without wasting extra time. Other than the direction of the
; data movement, this code is virtually identical to the above.

                mov     ax, DataSeg3
                mov     ds, ax
                mov     ax, DataSeg1
                mov     es, ax

                mov     cl, 127
                lea     si, Data1a+202h
iloop2:         mov     ch, 254/2
jloop2:         mov     dx, [si]
                mov     bx, [si-200h]
                mov     ax, dx
                shl     dx, 3
                add     bx, [si-1feh]
                mov     bp, [si+2]
                add     bx, [si+200h]
                add     dx, bp
                add     bx, [si+202h]
                add     dx, [si-202h]
                mov     di, [si-1fch]
                add     dx, [si-2]
                add     di, [si+4]
                add     dx, [si+1feh]
                add     di, [si+204h]
                shl     bp, 3
                add     dx, bx
                add     bp, ax
                shr     dx, 4
                add     bp, bx
                mov     es:[si], dx
                add     bp, di
                add     si, 4
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp
```

```
                jne     jloop2

                add     si, 4

                dec     cl
                jne     iloop2


                mov     ax, DataSeg4
                sub     ax, 40h
                mov     ds, ax
                mov     ax, DataSeg2
                sub     ax, 40h
                mov     es, ax

                mov     cl, 251-127-1
                mov     si, 202h
iloop3:         mov     ch, 254/2
jloop3:         mov     dx, [si]
                mov     bx, [si-200h]
                mov     ax, dx
                shl     dx, 3
                add     bx, [si-1feh]
                mov     bp, [si+2]
                add     bx, [si+200h]
                add     dx, bp
                add     bx, [si+202h]
                add     dx, [si-202h]
                mov     di, [si-1fch]
                add     dx, [si-2]
                add     di, [si+4]
                add     dx, [si+1feh]
                add     di, [si+204h]
                shl     bp, 3
                add     dx, bx
                add     bp, ax
                shr     dx, 4
                add     bp, bx
                mov     es:[si], dx
                add     bp, di
                add     si, 4
                shr     bp, 4
                dec     ch
                mov     es:[si-2], bp
                jne     jloop3

                add     si, 4

                dec     cl
                jne     iloop3

                mov     ax, dseg
                mov     ds, ax
                assume  ds:dseg

                dec     Iterations
                je      Done2
                jmp     hloop

Done2:          mov     ax, DataSeg1
                mov     bx, DataSeg2
                jmp     Finish

Done0:          mov     ax, DataSeg3
                mov     bx, DataSeg4
Finish:         mov     ds, ax
                print
                byte    "Writing result",cr,lf,0

; Convert data back to byte form and write to the output file:

                mov     ax, dseg
                mov     es, ax
```

```
                mov     cx, 32768
                lea     di, ImgData
                xor     si, si          ;Output data is at offset zero.
CopyLoop3:      lodsw                   ;Read a word from final array.
                stosb                   ;Write a byte to output array.
                dec     cx
                jne     CopyLoop3

                mov     ds, bx
                mov     cx, (251*256) - 32768
                xor     si, si
CopyLoop4:      lodsw                   ;Read final data word.
                stosb                   ;Write data byte to output array.
                dec     cx
                jne     CopyLoop4


; Okay, write the data to the output file:

                mov     ah, 3ch         ;Create output file.
                mov     cx, 0           ;Normal file attributes.
                mov     dx, dseg
                mov     ds, dx
                lea     dx, OutName
                int     21h
                jnc     GoodCreate
                print
                byte    "Could not create output file.",cr,lf,0
                jmp     Quit

GoodCreate:     mov     bx, ax          ;File handle.
                push    bx
                mov     dx, dseg        ;Where the data can be found.
                mov     ds, dx
                lea     dx, ImgData
                mov     cx, 256*251     ;Size of data file to write.
                mov     ah, 40h         ;Write operation.
                int     21h
                pop     bx              ;Retrieve handle for close.
                cmp     ax, 256*251     ;See if we wrote the data.
                je      GoodWrite
                print
                byte    "Did not write the file properly",cr,lf,0
                jmp     Quit

GoodWrite:      mov     ah, 3eh         ;Close operation.
                int     21h


Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

Of course, the absolute best way to improve the performance of any piece of code is with a better algorithm. All of the above assembly language versions were limited by a single requirement – they all must produce the same output file as the original Pascal program. Often, programmers lose sight of what it is that they are trying to accomplish and get so caught up in the computations they are performing that they fail to see other possibilities. The optimization example above is a perfect example. The assembly code faithfully preserves the semantics of the original Pascal program; it computes the weighted average

of all interior pixels as the sum of the eight neighbors around a pixel plus eight times the current pixel's value, with the entire sum divided by 16. Now this is a *good* blurring function, but it is not the *only* blurring function. A Photoshop (or other image processing program) user doesn't care about algorithms or such. When that user selects "blur image" they want it to go out of focus. Exactly how much out of focus is generally immaterial. In fact, the less the better because the user can always run the blur algorithm again (or specify some number of iterations). The following assembly language program shows how to get better performance by modifying the blurring algorithm to reduce the number of instructions it needs to execute in the innermost loops. It computes blurring by averaging a pixel with the four neighbors above, below, to the left, and to the right of the current pixel. This modification yields a program that runs 100 iterations in 2.2 seconds, a 12% improvement over the previous version:

```
; IMGPRCS.ASM
;
; An image processing program (Fifth optimization pass).
;
; This program blurs an eight-bit grayscale image by averaging a pixel
; in the image with the eight pixels around it. The average is computed
; by (CurCell*8 + other 8 cells)/16, weighting the current cell by 50%.
;
; Because of the size of the image (almost 64K), the input and output
; matrices are in different segments.
;
; Version #1: Straight-forward translation from Pascal to Assembly.
;
; Version #2: Three major optimizations. (1) used movsd instruction rather
;        than a loop to copy data from DataOut back to DataIn.
;        (2) Used repeat..until forms for all loops. (3) unrolled
;        the innermost two loops (which is responsible for most of
;        the performance improvement).
;
; Version #3: Used registers for all variables. Set up segment registers
;        once and for all through the execution of the main loop so
;        the code didn't have to reload ds each time through. Computed
;        index into each row only once (outside the j loop).
;
; Version #4: Eliminated copying data from DataOut to DataIn on each pass.
;        Removed hazards. Maintained common subexpressions. Did some
;        more loop unrolling.
;
; Version #6: Changed the blurring algorithm to use fewer computations.
;        This version does *NOT* produce the same data as the other
;        programs.
;
;
;       Performance comparisons (66 MHz 80486 DX/2 system, 100 iterations).
;
;       This code-                    2.2 seconds.
;       3rd optmization pass-         2.5 seconds.
;       2nd optimization pass-        4 seconds.
;       1st optimization pass-        6 seconds.
;       Original ASM code-            36 seconds.

;       «Lots of deleted code here, see the original program»


                print
                byte        "Computing Result",cr,lf,0


                assume      ds:InSeg, es:OutSeg

                mov         ax, InSeg
                mov         ds, ax
                mov         ax, OutSeg
                mov         es, ax

; Copy the data once so we get the edges in both arrays.

                mov         cx, (251*256)/4
                lea         si, DataIn
```

```
                lea     di, DataOut
        rep     movsd


; “hloop” repeats once for each iteration.

hloop:
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax

; “iloop” processes the rows in the matrices.

                mov     cl, 249
iloop:          mov     bh, cl          ;i*256
                mov     bl, 1           ;Start at j=1.
                mov     ch, 254/2       ;# of times through loop.
                mov     si, bx
                mov     dh, 0           ;Compute sum here.
                mov     bh, 0
                mov     ah, 0

; “jloop” processes the individual elements of the array.
; This loop has been unrolled once to allow the two portions to share
; some common computations.

jloop:

; The sum of DataIn [i-1][j] + DataIn[i-1][j+1] + DataIn[i+1][j] +
; DataIn [i+1][j+1] will be used in the second half of this computation.
; So save its value in a register (di) until we need it again.

                mov     dl, DataIn[si]                  ;[i,j]
                mov     al, DataIn[si-256]              ;[I-1,j]
                shl     dx, 2                           ;[i,j]*4
                mov     bl, DataIn[si-1]                ;[i,j-1]
                add     dx, ax
                mov     al, DataIn[si+1]                ;[i,j+1]
                add     dx, bx
                mov     bl, DataIn[si+256]              ;[i+1,j]
                add     dx, ax
                shl     ax, 2                           ;[i,j+1]*4
                add     dx, bx
                mov     bl, DataIn[si-255]              ;[i-1,j+1]
                shr     dx, 3                           ;Divide by 8.
                add     ax, bx
                mov     DataOut[si], dl
                mov     bl, DataIn[si+2]                ;[i,j+2]
                mov     dl, DataIn[si+257]              ;[i+1,j+1]
                add     ax, bx
                mov     bl, DataIn[si]                  ;[i,j]
                add     ax, dx
                add     ax, bx
                shr     ax, 3
                dec     ch
                mov     DataOut[si+1], al
                jne     jloop

                dec     cl
                jne     iloop

                dec     bp
                je      Done


; Special case so we don’t have to move the data between the two arrays.
; This is an unrolled version of the hloop that swaps the input and output
; arrays so we don’t have to move data around in memory.

                mov     ax, OutSeg
                mov     ds, ax
                mov     ax, InSeg
                mov     es, ax
```

```
                assume  es:InSeg, ds:OutSeg

hloop2:

                mov     cl, 249
iloop2:         mov     bh, cl
                mov     bl, 1
                mov     ch, 254/2
                mov     si, bx
                mov     dh, 0
                mov     bh, 0
                mov     ah, 0
jloop2:
                mov     dl, DataOut[si-256]
                mov     al, DataOut[si-255]
                mov     bl, DataOut[si+257]
                add     dx, ax
                mov     al, DataOut[si+256]
                add     dx, bx
                mov     bl, DataOut[si+1]
                add     dx, ax
                mov     al, DataOut[si+255]

                mov     di, dx

                add     dx, bx
                mov     bl, DataOut[si-1]
                add     dx, ax
                mov     al, DataOut[si]
                add     dx, bx
                mov     bl, DataOut[si-257]
                shl     ax, 3
                add     dx, bx
                add     dx, ax
                shr     ax, 3
                shr     dx, 4
                mov     DataIn[si], dl

                mov     dx, di
                mov     bl, DataOut[si-254]
                add     dx, ax
                mov     al, DataOut[si+2]
                add     dx, bx
                mov     bl, DataOut[si+258]
                add     dx, ax
                mov     al, DataOut[si+1]
                add     dx, bx
                shl     ax, 3
                add     si, 2
                add     dx, ax
                mov     ah, 0
                shr     dx, 4
                dec     ch
                mov     DataIn[si-1], dl
                jne     jloop2

                dec     cl
                jne     iloop2

                dec     bp
                je      Done2
                jmp     hloop


; Kludge to guarantee that the data always resides in the output segment.

Done2:
                mov     ax, InSeg
                mov     ds, ax
                mov     ax, OutSeg
                mov     es, ax
                mov     cx, (251*256)/4
                lea     si, DataIn
                lea     di, DataOut
```

```
        rep   movsd

Done:         print
              byte      "Writing result",cr,lf,0


;       «Lots of delete code here, see the original program»
```

One very important thing to keep in mind about the codein this section is that we've optimized it for 100 iterations. While it turns out that these optimizations apply equally well to more iterations, this isn't necessarily true for fewer iterations. In particular, if we run only one iteration, any copying of data at the end of the operation will easily consume a large part of the time we save by the optimizations. Since it is very rare for a user to blur an image 100 times in a row, our optimizations may not be as good as we could make them. However, this section does provide a good example of the steps you must go through in order to optimize a given program. One hundred iterations was a good choice for this example because it was easy to measure the running time of all versions of the program. However, you must keep in mind that you should optimize your programs for the expected case, not an arbitrary case.

25.6 Summary

Computer software often runs significantly slower than the task requires. The process of increasing the speed of a program is known as *optimization*. Unfortunately, optimization is a difficult and time-consuming task, something not to be taken lightly. Many programmers often optimize their programs before they've determined that there is a need to do so, or (worse yet) they optimize a portion of a program only to find that they have to rewrite that code after they've optimized it. Others, out of ignorance, often wind up optimizing the wrong sections of their programs. Since optimization is a slow and difficult process, you want to try and make sure you only optimize your code *once*. This suggests that optimization should be your last task when writing a program.

One school of thought that completely embraces this philosophy is the *Optimize Late* group. Their arguement is that program optimization often destroys the readability and maintanability of a program. Therefore, one should only take this step when absolutely necessary and only at the end of the program development stage.

The *Optimize Early* crowd knows, from experience, that programs that are not written to be fast often need to be completely rewritten to make them fast. Therefore, they often take the attitude that optimization should take place along with normal program development. Generally, the optimize early group's view of optimization is typically far different from the optimize late group. The optimize early group claims that the extra time spent optimizing a program during development requires less time than developing a program and then optimizing it. For all the details on this *religious* battle, see

- "When to Optimize, When Not to Optimize" on page 1311

After you've written a program and determine that it runs too slowly, the next step is to locate the code that runs too slow. After identifying the slow sections of your program, you can work on speeding up your programs. Locating that 10% of the code that requires 90% of the execution time is not always an easy task. The four common techniques people use are trial and error, optimize everything, program analysis, and experimental analysis (i.e., use a profiler). Finding the "hot spots" in a program is the first optimization step. To learn about these four techniques, see

- "How Do You Find the Slow Code in Your Programs?" on page 1313

A convincing arguement the optimize late folks use is that machines are so fast that optimization is rarely necessary. While this arguement is often overstated, it is often true that many unoptimized programs run fast enough and do not require any optimization for satisfactory performance. On the other hand, programs that run fine by themselves may be too slow when running concurrently with other software. To see the strengths and weaknesses of this arguement, see

- “Is Optimization Necessary?” on page 1314

There are three forms of optimization you can use to improve the performance of a program: choose a better algorithm, choose a better implementation of an algorithm, or “count cycles.” Many people (especially the optimize late crowd) only consider this last case “optimization.” This is a shame, because the last case often produces the smallest incremental improvement in performance. To understand these three forms of optimization, see

- “The Three Types of Optimization” on page 1315

Optimization is not something you can learn from a book. It takes lots of experience and practice. Unfortunately, those with little practical experience find that their efforts rarely pay off well and generally assume that optimization is not worth the trouble. The truth is, they do not have sufficient experience to write truly optimal code and their frustration prevents them from gaining such experience. The latter part of this chapter devotes itself to demonstrating what one can achieve when optimizing a program. Always keep this example in mind when you feel frustrated and are beginning to believe you cannot improve the performance of your program. For details on this example, see

- “Improving the Implementation of an Algorithm” on page 1317

Appendix B: Annotated Bibliography

There are a wide variety of texts available for those who are interested in learning more about assembly language or other topics this text covers. The following is a partial list of texts that may be of interest to you. Many of these texts are now out of print. Please consult your local library if you cannot find a particular text at a bookstore.

Microprocessor Programming for Computer Hobbyists
Neill Graham
TAB books
ISBN 0-8306-6952-3
1977
This book provides a gentle introduction to data structures for computer hobbyists. Although it uses the PL/M programming language, many of the concepts apply directly to assembly language programs.

IBM Assembler Language and Programming
Peter Able
Prentice-Hall
ISBN 0-13-448143-7
1987
A college text book on assembly language. Contains good sections on DOS and disk formats for earlier versions of DOS.

MS-DOS Developer's Guide
John Angermeyer and Keven Jaeger
Howard W. Sams & Co.
ISBN 0-672-22409-7
An excellent reference book on programming MS-DOS.

Compilers: Principles, Techniques, and Tools
Alfred Aho, Ravi Sethi, and Jeffrey Ullman
Addison Wesley
ISBN 0-201-10088-6
1986
The standard text on compiler design and implementation. Contains lots of material on pattern matching and other related subjects.

C Programmer's Guide to Serial Communications
Joe Campbell
Howard W. Sams & Co.
ISBN 0-672-22584-0
An indispensible guide to serial communications. Although written specifically for C programmers, the material applies equally well to assembly language programmers.

The MS-DOS Encyclopedia
Ray Duncan, General Editor & various authors
Microsoft Press
ISBN 1-55615-049-0
An excellent description of MS-DOS programming. Contains especially good sections on resident programs and device drivers. Quite expensive, but well worth it.

Zen of Assembly Language
Michael Abrash
Scott Foresman
ISBN 0-673-38602-3
1990
The first really great book on 80x86 code optimization. There are only two things wrong with this book. (1) It is out of print. (2) The optimization techniques apply mostly to the 8088 and 80286 processors, they do not apply as well to the 80386 and later processors. That's okay, see the next entry below.

Zen of Code Optimization
Michael Abrash
Coriolis Group Books
ISBN 1-883577-03-9
1994
Here is Michael Abrash's book updated for the 80386, 80486, and Pentium processors. An absolute must-have for 80x86 assembly language programmers.

Assembler Inside & Out
Harley Hahn
McGraw-Hill
ISBN 0-07-881842-7
1992
A reasonable 80x86 assembly language text. This one is notable because Microsoft ships this text with every copy of MASM.

Assembly Language Subroutines for MS-DOS (2nd Edition)
Leo J. Scanlon
Windcrest
ISBN 0-8306-7649-X
This book is full of little code examples. The routines themselves are not earth-shaking, but it does provide lots of good code examples for those individuals who learn by example.

Advanced Assembly Language
Steven Holzner
Brady/Peter Norton
ISBN 0-13-658774-7
1991
This book provides a basic introduction to programming many of the PC's hardware devices in assembly language. Despite its name, it is not truly an *advanced* assembly language programming text.

Assembly Language. For Real Programmers Only.
Marcus Johnson
Sams Publishing
ISBN 0-672-48470
A comprehensive book (over 1,300 pages) with lots of example code.

The Revolutionary Guide to Assembly Language
Vitaly Maljugin, Jacov Izrailevich, Semyon Lavin, and Alksandr Sopin
Wrox Press
ISBN 1-874416-12-5
1993
Another comprehensive text on assembly language. This one spends considerable time discussing the PC's hardware. This text also includes sections on how to interface assembly language with the Clipper (dBase compiler) programming language.

The Waite Group's Microsoft Macro Assembler Bible
Nabajyoti Barkakati and Randall Hyde
Sams
ISBN 0-672-30155-5
1992
A comprehensive reference manual to MASM 6.x and the 8088 through the 80486.

Computer Organization & Design: The Hardware/Software Interface
David Patterson and John Hennessy
Morgan Kaufmann Publishers
ISBN 1-55860-223-2
1993
An excellent text on machine organization, one of the best in the field.

Computer Architecture, A Quantitative Approach
John Hennessy and David Patternson
Morgan Kaufmann Publishers
ISBN 1-55860-069-8
1990
One of the standard texts on computer architecture. Although it emphasizes RISC processors over CISC, many of the topics discussed apply to superscalar and pipelined CISC processors as well.

IBM Microcomputers: A Programmer's Handbook
Julio Sanchez and Maria P. Canton
McGraw Hill
ISBN 0-07--54594-4
1990
One of the best reference manuals covering the PC's hardware. An absolute must-have book for those interested in programming peripheral devices on the PC.

The Undocumented PC
Frank Van Gilluwe
Addison Wesley
ISBN 0-201-62277-7
1994
Another excellent text that covers the PC's hardware and how to program peripheral devices.

The Indispensible PC Hardware Book
Hans-Peter Messmer
Addison Wesley
ISBN 0-201-62424-9
Yet another great PC hardware book. This one even describes the low-level operation of various silicon devices in a way even beginners can understand. It also provides an excellent hardware reference guide to the 80386 and 80486 microprocessor chips.

Programmer's Technical Reference: The Processor and Coprocessor
Robert L. Hummel
Ziff-Davis Press
ISBN 1-56276-016-5
1992
One of the premier references on the 80x86 family from the 8088 through the 80486 chips. Also provides an excellent discussion of the 8087, 80287, 80387, and 487 math coprocessors.

Microsoft MS-DOS Programmer's Reference
Written by Microsoft Corporation
Microsoft Press
ISBN 1-55615-329-5
1991
The official guide to programming MS-DOS, directly from Microsoft.

Undocumented DOS. A Programmer's Guide to Reserved MS-DOS Functions and Data Structures
Andrew Schulman, Raymond Michels, Jim Kyle, Tim Patterson, David Maxey, and Ralf Brown
Addison Wesley
ISBN 0-201-57064-5
1990
This book describes lots of features available to MS-DOS that Microsoft never bothered to document. This text contains vital information to TSR and protected mode programmers.

Introduction to Automata Theory, Languages, and Computation
John Hopcroft and Jeffrey Ullman
Addison Wesley
1979
ISBN 0-201-02988-X
Very concise, but one of the standard texts on automata theory, pattern matching, and computability.

The Art of Computer Programming,
Vol 1: Fundamental Algorithms
Vol 2: Seminumerical Algorithms
Vol 3: Sorting and Searching
Donald Knuth
Addison Wesley
1973
One of the finest sets of text on data structures and algorithms available for assembly language programmers. Donald Knuth uses a hypothetical assembly language, *MIX*, to present most algorithms. Code in these texts is very easy to convert to 80x86 assembly language.

Appendix C: Keyboard Scan Codes

Table 90: PC Keyboard Scan Codes (in hex)

Key	Down	Up	Key	Down	Up	Key	Down	Up	Key	Down	Up
Esc	1	81	[{	1A	9A	, <	33	B3	***center***	4C	CC
1 !	2	82	] }	1B	9B	. >	34	B4	***right***	4D	CD
2 @	3	83	Enter	1C	9C	/ ?	35	B5	+	4E	CE
3 #	4	84	Ctrl	1D	9D	R shift	36	B6	***end***	4F	CF
4 $	5	85	A	1E	9E	* PrtSc	37	B7	***down***	50	D0
5 %	6	86	S	1F	9F	alt	38	B8	***pgdn***	51	D1
6 ^	7	87	D	20	A0	space	39	B9	***ins***	52	D2
7 &	8	88	F	21	A1	CAPS	3A	BA	***del***	53	D3
8 *	9	89	G	22	A2	F1	3B	BB	/	E0 35	B5
9 (	0A	8A	H	23	A3	F2	3C	BC	***enter***	E0 1C	9C
0)	0B	8B	J	24	A4	F3	3D	BD	F11	57	D7
- _	0C	8C	K	25	A5	F4	3E	BE	F12	58	D8
= +	0D	8D	L	26	A6	F5	3F	BF	ins	E0 52	D2
Bksp	0E	8E	; :	27	A7	F6	40	C0	del	E0 53	D3
Tab	0F	8F	‘ “	28	A8	F7	41	C1	home	E0 47	C7
Q	10	90	` ~	29	A9	F8	42	C2	end	E0 4F	CF
W	11	91	L shift	2A	AA	F9	43	C3	pgup	E0 49	C9
E	12	92	\ |	2B	AB	F10	44	C4	pgdn	E0 51	D1
R	13	93	Z	2C	AC	NUM	45	C5	left	E0 4B	CB
T	14	94	X	2D	AD	SCRL	46	C6	right	E0 4D	CD
Y	15	95	C	2E	AE	***home***	47	C7	up	E0 48	C8
U	16	96	V	2F	AF	***up***	48	C8	down	E0 50	D0
I	17	97	B	30	B0	***pgup***	49	C9	R alt	E0 38	B8
O	18	98	N	31	B1	-	4A	CA	R ctrl	E0 1D	9D
P	19	99	M	32	B2	***left***	4B	CB	Pause	E1 1D 45 E1 9D C5	-

Table 91: Keyboard Codes (in hex)

Key	Scan Code	ASCII	Shift[a]	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num
Esc	01	1B	1B	1B		1B	1B	1B	1B
1 !	02	31	21		***7800***	31	31	31	31
2 @	03	32	40	***0300***	***7900***	32	32	32	32
3 #	04	33	23		***7A00***	33	33	33	33
4 $	05	34	24		***7B00***	34	34	34	34
5 %	06	35	25		***7C00***	35	35	35	35
6 ^	07	36	5E	1E	***7D00***	36	36	36	36
7 &	08	37	26		***7E00***	37	37	37	37
8 *	09	38	2A		***7F00***	38	38	38	38
9 (	0A	39	28		***8000***	39	39	39	39
0)	0B	30	29		***8100***	30	30	30	30
- _	0C	2D	5F	1F	***8200***	2D	2D	5F	5F
= +	0D	3D	2B		***8300***	3D	3D	2B	2B
Bksp	0E	08	08	7F		08	08	08	08
Tab	0F	09	***0F00***			09	09	***0F00***	***0F00***
Q	10	71	51	11	***1000***	71	51	71	51
W	11	77	57	17	***1100***	77	57	77	57
E	12	65	45	05	***1200***	65	45	65	45
R	13	72	52	12	***1300***	72	52	72	52
T	14	74	54	14	***1400***	74	54	74	54
Y	15	79	59	19	***1500***	79	59	79	59
U	16	75	55	15	***1600***	75	55	75	55
I	17	69	49	09	***1700***	69	49	69	49
O	18	6F	4F	0F	***1800***	6F	4F	6F	4F
P	19	70	50	10	***1900***	70	50	70	50
[{	1A	5B	7B	1B		5B	5B	7B	7B
] }	1B	5D	7D	1D		5D	5D	7D	7D
enter	1C	0D	0D	0A		0D	0D	0A	0A
ctrl	1D								
A	1E	61	41	01	***1E00***	61	41	61	41
S	1F	73	53	13	***1F00***	73	53	73	53
D	20	64	44	04	***2000***	64	44	64	44
F	21	66	46	06	***2100***	66	46	66	46
G	22	67	47	07	***2200***	67	47	67	47
H	23	68	48	08	***2300***	68	48	68	48
J	24	6A	4A	0A	***2400***	6A	4A	6A	4A
K	25	6B	4B	0B	***2500***	6B	4B	6B	4B
L	26	6C	4C	0C	***2600***	6C	4C	6C	4C
; :	27	3B	3A			3B	3B	3A	3A
‘ “	28	27	22			27	27	22	22
` ~	29	60	7E			60	60	7E	7E
Lshift	2A								
\ |	2B	5C	7C	1C		5C	5C	7C	7C
Z	2C	7A	5A	1A	***2C00***	7A	5A	7A	5A
X	2D	78	58	18	***2D00***	78	58	78	58
C	2E	63	43	03	***2E00***	63	43	63	43
V	2F	76	56	16	***2F00***	76	56	76	56
B	30	62	42	02	***3000***	62	42	62	42
Key	Scan Code	ASCII	Shift	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num

Table 91: Keyboard Codes (in hex)

Key	Scan Code	ASCII	Shift[a]	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num
N	31	6E	4E	0E	***3100***	6E	4E	6E	4E
M	32	6D	4D	0D	***3200***	6D	4D	6D	4D
, <	33	2C	3C			2C	2C	3C	3C
. >	34	2E	3E			2E	2E	3E	3E
/ ?	35	2F	3F			2F	2F	3F	3F
Rshift	36								
* PrtSc	37	2A	INT 5[b]	10[c]		2A	2A	INT 5	INT 5
alt	38								
space	39	20	20	20		20	20	20	20
caps	3A								
F1	3B	***3B00***	***5400***	***5E00***	***6800***	***3B00***	***3B00***	***5400***	***5400***
F2	3C	***3C00***	***5500***	***5F00***	***6900***	***3C00***	***3C00***	***5500***	***5500***
F3	3D	***3D00***	***5600***	***6000***	***6A00***	***3D00***	***3D00***	***5600***	***5600***
F4	3E	***3E00***	***5700***	***6100***	***6B00***	***3E00***	***3E00***	***5700***	***5700***
F5	3F	***3F00***	***5800***	***6200***	***6C00***	***3F00***	***3F00***	***5800***	***5800***
F6	40	***4000***	***5900***	***6300***	***6D00***	***4000***	***4000***	***5900***	***5900***
F7	41	***4100***	***5A00***	***6400***	***6E00***	***4100***	***4100***	***5A00***	***5A00***
F8	42	***4200***	***5B00***	***6500***	***6F00***	***4200***	***4200***	***5B00***	***5B00***
F9	43	***4300***	***5C00***	***6600***	***7000***	***4300***	***4300***	***5C00***	***5C00***
F10	44	***4400***	***5D00***	***6700***	***7100***	***4400***	***4400***	***5D00***	***5D00***
num	45								
scrl	46								
home	47	***4700***	37	***7700***		37	4700	37	4700
up	48	***4800***	38			38	4800	38	4800
pgup	49	***4900***	39	***8400***		39	4900	39	4900
-[d]	4A	2D	2D			2D	2D	2D	2D
left	4B	***4B00***	34	***7300***		34	4B00	34	4B00
center	4C	***4C00***	35			35	4C00	35	4C00
right	4D	***4D00***	36	***7400***		36	4D00	36	4D00
+[e]	4E	2B	2B			2B	2B	2B	2B
end	4F	***4F00***	31	***7500***		31	4F00	31	4F00
down	50	***5000***	32			32	5000	32	5000
pgdn	51	***5100***	33	***7600***		33	5100	33	5100
ins	52	***5200***	30			30	5200	30	5200
del	53	***5300***	2E			2E	5300	2E	5300
Key	Scan Code	ASCII	Shift	Ctrl	Alt	Num	Caps	Shift Caps	Shift Num

a. For the alphabetic characters, if capslock is active then see the shift-capslock column.
b. Pressing the PrtSc key does not produce a scan code. Instead, BIOS executes an int 5 instruction which should print the screen.
c. This is the control-P character that will activate the printer under MS-DOS.
d. This is the minus key on the keypad.
e. This is the plus key on the keypad.

Table 92: Keyboard Related BIOS Variables

Name	Address[a]	Size	Description
KbdFlags1 (modifier flags)	40:17	Byte	This byte maintains the current status of the modifier keys on the keyboard. The bits have the following meanings: bit 7: Insert mode toggle bit 6: Capslock toggle (1=capslock on) bit 5: Numlock toggle (1=numlock on) bit 4: Scroll lock toggle (1=scroll lock on) bit 3: Alt key (1=alt is down) bit 2: Ctrl key (1=ctrl is down) bit 1: Left shift key (1=left shift is down) bit 0: Right shift key (1=right shift is down)
KbdFlags2 (Toggle keys down)	40:18	Byte	Specifies if a toggle key is currently down. bit 7: Insert key (currently down if 1) bit 6: Capslock key (currently down if 1) bit 5: Numlock key (currently down if 1) bit 4: Scroll lock key (currently down if 1) bit 3: Pause state locked (ctrl-Numlock) if one bit 2: SysReq key (currently down if 1) bit 1: Left alt key (currently down if 1) bit 0: Left ctrl key (currently down if 1)
AltKpd	40:19	Byte	BIOS uses this to compute the ASCII code for an alt-Keypad sequence.
BufStart	40:80	Word	Offset of start of keyboard buffer (1Eh). Note: this variable is not supported on many systems, be careful if you use it.
BufEnd	40:82	Word	Offset of end of keyboard buffer (3Eh). See the note above.
KbdFlags3	40:96	Byte	Miscellaneous keyboard flags. bit 7: Read of keyboard ID in progress bit 6: Last char is first kbd ID character bit 5: Force numlock on reset bit 4: 1 if 101-key kbd, 0 if 83/84 key kbd. bit 3: Right alt key pressed if 1 bit 2: Right ctrl key pressed if 1 bit 1: Last scan code was E0h bit 0: Last scan code was E1h
KbdFlags4	40:97	Byte	More miscellaneous keyboard flags. bit 7: Keyboard transmit error bit 6: Mode indicator update bit 5: Resend receive flag bit 4: Acknowledge received bit 3: Must always be zero bit 2: Capslock LED (1=on) bit 1: Numlock LED (1=on) bit 0: Scroll lock LED (1=on)

a. Addresses are all given in hexadecimal

Table 93: On-Board Keyboard Controller Commands (Port 64h)

Value (hex)	Description
20	Transmit keyboard controller's command byte to system as a scan code at port 60h.
60	The next byte written to port 60h will be stored in the keyboard controller's command byte.
A4	Test if a password is installed (PS/2 only). Result comes back in port 60h. 0FAh means a password is installed, 0F1h means no password.
A5	Transmit password (PS/2 only). Starts receipt of password. The next sequence of scan codes written to port 60h, ending with a zero byte, are the new password.
A6	Password match. Characters from the keyboard are compared to password until a match occurs.
A7	Disable mouse device (PS/2 only). Identical to setting bit five of the command byte.
A8	Enable mouse device (PS/2 only). Identical to clearing bit five of the command byte.
A9	Test mouse device. Returns 0 if okay, 1 or 2 if there is a stuck clock, 3 or 4 if there is a stuck data line. Results come back in port 60h.
AA	Initiates self-test. Returns 55h in port 60h if successful.
AB	Keyboard interface test. Tests the keyboard interface. Returns 0 if okay, 1 or 2 if there is a stuck clock, 3 or 4 if there is a stuck data line. Results come back in port 60h.
AC	Diagnostic. Returns 16 bytes from the keyboard's microcontroller chip. Not available on PS/2 systems.
AD	Disable keyboard. Same operation as setting bit four of the command register.
AE	Enable keyboard. Same operation as clearing bit four of the command register.
C0	Read keyboard input port to port 60h. This input port contains the following values: bit 7: Keyboard inhibit keyswitch (0 = inhibit, 1 = enabled). bit 6: Display switch (0=color, 1=mono). bit 5: Manufacturing jumper. bit 4: System board RAM (always 1). bits 0-3: undefined.
C1	Copy input port (above) bits 0-3 to status bits 4-7. (PS/2 only)
C2	Copy input pot (above) bits 4-7 to status port bits 4-7. (PS/2 only).
D0	Copy microcontroller output port value to port 60h (see definition below).
D1	Write the next data byte written to port 60h to the microcontroller output port. This port has the following definition: bit 7: Keyboard data. bit 6: Keyboard clock. bit 5: Input buffer empty flag. bit 4: Output buffer full flag. bit 3: Undefined. bit 2: Undefined. bit 1: Gate A20 line. bit 0: System reset (if zero). Note: writing a zero to bit zero will reset the machine. Writing a one to bit one combines address lines 19 and 20 on the PC's address bus.

Table 93: On-Board Keyboard Controller Commands (Port 64h)

Value (hex)	Description
D2	Write keyboard buffer. The keyboard controller returns the next value sent to port 60h as though a keypress produced that value. (PS/2 only).
D3	Write mouse buffer. The keyboard controller returns the next value sent to port 60h as though a mouse operation produced that value. (PS/2 only).
D4	Writes the next data byte (60h) to the mouse (auxiliary) device. (PS/2 only).
E0	Read test inputs. Returns in port 60h the status of the keyboard serial lines. Bit zero contains the keyboard clock input, bit one contains the keyboard data input.
F*x*	Pulse output port (see definition for D1). Bits 0-3 of the keyboard controller command byte are pulsed onto the output port. Resets the system if bit zero is a zero.

Table 94: Keyboard to System Transmissions

Value (hex)	Description
00	Data overrun. System sends a zero byte as the last value when the keyboard controller's internal buffer overflows.
1..58 81..D8	Scan codes for key presses. The positive values are down codes, the negative values (H.O. bit set) are up codes.
83AB	Keyboard ID code returned in response to the F2 command (PS/2 only).
AA	Returned during basic assurance test after reset. Also the up code for the left shift key.
EE	Returned by the ECHO command.
F0	Prefix to certain up codes (N/A on PS/2).
FA	Keyboard acknowledge to keyboard commands other than resend or ECHO.
FC	Basic assurance test failed (PS/2 only).
FD	Diagnostic failure (not available on PS/2).
FE	Resend. Keyboard requests the system to resend the last command.
FF	Key error (PS/2 only).

Table 95: Keyboard Microcontroller Commands (Port 60h)

Value (hex)	Description
ED	Send LED bits. The next byte written to port 60h updates the LEDs on the keyboard. The parameter (next) byte contains: bits 3-7: Must be zero. bit 2: Capslock LED (1 = on, 0 = off). bit 1: Numlock LED (1 = on, 0 = off). bit 0: Scroll lock LED (1 = on, 0 = off).
EE	Echo commands. Returns 0EEh in port 60h as a diagnostic aid.
F0	Select alternate scan code set (PS/2 only). The next byte written to port 60h selects one of the following options: 00: Report current scan code set in use (next value read from port 60h). 01: Select scan code set #1 (standard PC/AT scan code set). 02: Select scan code set #2. 03: Select scan code set #3.
F2	Send two-byte keyboard ID code as the next two bytes read from port 60h (PS/2 only).
F3	Set Autorepeat delay and repeat rate. Next byte written to port 60h determines rate: bit 7: must be zero bits 5,6: Delay. 00- $^1/_4$ sec, 01- $^1/_2$ sec, 10- $^3/_4$ sec, 11- 1 sec. bits 0-4: Repeat rate. 0- approx 30 chars/sec to 1Fh- approx 2 chars/sec.
F4	Enable keyboard.
F5	Reset to power on condition and wait for enable command.
F6	Reset to power on condition and begin scanning keyboard.
F7	Make all keys autorepeat (PS/2 only).
F8	Set all keys to generate an up code and a down code (PS/2 only).
F9	Set all keys to generate an up code only (PS/2 only).
FA	Set all keys to autorepeat and generate up and down codes (PS/2 only).
FB	Set an individual key to autorepeat. Next byte contains the scan code of the desired key. (PS/2 only).
FC	Set an individual key to generate up and down codes. Next byte contains the scan code of the desired key. (PS/2 only).
FD	Set an individual key to generate only down codes. Next byte contains the scan code of the desired key. (PS/2 only).
FE	Resend last result. Use this command if there is an error receiving data.
FF	Reset keyboard to power on state and start the self-test.

Table 96: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
0		al- ASCII character ah- scan code	Read character. Reads next available character from the system's type ahead buffer. Wait for a keystroke if the buffer is empty.
1		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Checks to see if a character is available in the type ahead buffer. Sets the zero flag if not key is available, clears the zero flag if a key is available. If there is an available key, this function returns the ASCII and scan code value in ax. The value in ax is undefined if no key is available.
2		al- shift flags	Returns the current status of the shift flags in al. The shift flags are defined as follows: bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Alt key is down bit 2: Ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down
3	al = 5 bh = 0, 1, 2, 3 for 1/4, 1/2, 3/4, or 1 second delay bl= 0..1Fh for 30/sec to 2/sec.		Set auto repeat rate. The bh register contains the amount of time to wait before starting the autorepeat operation, the bl register contains the autorepeat rate.
5	ch = scan code cl = ASCII code		Store keycode in buffer. This function stores the value in the cx register at the end of the type ahead buffer. Note that the scan code in ch doesn't have to correspond to the ASCII code appearing in cl. This routine will simply insert the data you provide into the system type ahead buffer.
10h		al- ASCII character ah- scan code	Read extended character. Like ah=0 call, except this one passes all key codes, the ah=0 call throws away codes that are not PC/XT compatible.
11h		ZF- Set if no key. ZF- Clear if key available. al- ASCII code ah- scan code	Like the ah=01h call except this one does not throw away keycodes that are not PC/XT compatible (i.e., the extra keys found on the 101 key keyboard).

Table 96: BIOS Keyboard Support Functions

Function # (AH)	Input Parameters	Output Parameters	Description
12h		al- shift flags ah- extended shift flags	Returns the current status of the shift flags in ax. The shift flags are defined as follows: bit 15: SysReq key pressed bit 14: Capslock key currently down bit 13: Numlock key currently down bit 12: Scroll lock key currently down bit 11: Right alt key is down bit 10:Right ctrl key is down bit 9: Left alt key is down bit 8: Left ctrl key is down bit 7: Insert toggle bit 6: Capslock toggle bit 5: Numlock toggle bit 4: Scroll lock toggle bit 3: Either alt key is down (some machines, left only) bit 2: Either ctrl key is down bit 1: Left shift key is down bit 0: Right shift key is down

Appendix D: Instruction Set Reference

This section provides encodings and approximate cycle times for all instructions that you would normally execute in *real* mode on an Intel processor. Missing are the special instructions on the 80286 and later processors that manipulate page tables, segment descriptors, and other instructions that only an operating system should use. The cycle times are approximate. To determine exact execution times, you will need to run an experiment. The cycle times are given for comparison purposes only.

Key to special bits in encodings:

x:	Don't care. Can be zero or one.
s:	Sign extension bit for immediate operands. If zero, immediate operand is 16 or 32 bits depending on destination operand size. If s bit is one, then the immediate operand is eight bits and the CPU sign extends to 16 or 32 bits, as appropriate.
rrr:	Same as reg field in [mod-reg-r/m] byte.

Other Notes:

[disp]	This field can be zero, one, two, or four bytes long as required by the instruction.
[imm]	This field is one byte long if the operand is an eight bit operand or if the *s* bit in the instruction opcode is one. It is two or four bytes long if the *s* bit contains zero and the destination operand is 16 or 32 bits, respectively.
[mod-reg-r/m]:	Instructions that have a mod-reg-r/m byte may have a scaled index byte (sib) and a zero, one, two, or four byte displacement. See Appendix E for details concerning the encoding of this portion of the instruction.
reg,reg	Many instructions allow two operands using a [mod-reg-r/m] byte. A single *direction* bit in the opcode determines whether the instruction treats the *reg* operand as the destination or the mod-r/m operand as the destination (e.g., mov reg,mem vs. mov mem,reg). Such instructions also allow two register operands. It turns out there are two encodings for each such reg-reg instruction. That is, you can encode an instruction like mov ax, bx with ax encoded in the reg field and bx encoded in the mod-r/m field, or you can encode it with bx encoded in the reg field and ax encoded in the mod-r/m field. Such instructions always have an *x* bit in the opcode. If the *x* bit is zero, the destination is the register specified by the mod-r/m field. If the *x* bit is one, the destination is the register specified by the reg field. Other types of instructions support multiple encodings for similar reasons.

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
aaa	0011 0111	8	8	3	4	3	3
aad	1101 0101 0000 1010	60	60	14	19	14	10
aam	1101 0100 0000 1010	83	83	16	17	15	18
aas	0011 1111	8	8	3	4	3	3
adc reg8, reg8	0001 00x0 [11-reg-r/m]	3	‘3	2	2	1	1
adc reg16, reg16	0001 00x1 [11-reg-r/m]	3	3	2	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
adc reg32, reg32	0110 0110 0001 00x1 [11-reg-r/m]	3	3	2	2	1	1
adc reg8, mem8	0001 0010 [mod-reg-r/m]	9+EA	9+EA	7	6	2	2
adc reg16, mem16	0001 0011 [mod-reg-r/m]	13+EA	9+EA	7	6	2	2
adc reg32, mem32	0110 0110 0001 0011 [mod-reg-r/m]	-	-	-	6	2	2
adc mem8, reg8	0001 0000 [mod-reg-r/m]	16+EA	16+EA	7	7	3	3
adc mem16, reg16	0001 0001 [mod-reg-r/m]	24+EA	16+EA	7	7	3	3
adc mem32, reg32	0110 0110 0001 0001 [mod-reg-r/m]	-	-	-	7	3	3
adc reg8, imm8	1000 00x0 [11-010-r/m] [imm]	4	4	3	2	1	1
adc reg16, imm16	1000 00s0 [11-010-r/m] [imm]	4	4	3	2	1	1
adc reg32, imm32	0110 0110 1000 00s0 [11-010-r/m] [imm]	4	4	3	2	1	1
adc mem8, imm8	1000 00x0 [mod-010-r/m] [imm]	17+EA	17+EA	7	7	3	3
adc mem16, imm16	1000 00s1 [mod-010-r/m] [imm]	23+EA	17+EA	7	7	3	3
adc mem32, imm32	0110 0110 1000 00s1 [mod-010-r/m] [imm]	-	-	-	7	3	3
adc al, imm	0001 0100 [imm]	4	4	3	2	1	1
adc ax, imm	0001 0101 [imm]	4	4	3	2	1	1
adc eax, imm	0110 0110 0001 0101 [imm]	-	-	-	2	1	1
add reg8, reg8	0000 00x0 [11-reg-r/m]	3	‘3	2	2	1	1
add reg16, reg16	0000 00x1 [11-reg-r/m]	3	3	2	2	1	1
add reg32, reg32	0110 0110 0000 00x1 [11-reg-r/m]	3	3	2	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
add reg8, mem8	0000 0010 [mod-reg-r/m]	9+EA	9+EA	7	6	2	2
add reg16, mem16	0000 0011 [mod-reg-r/m]	13+EA	9+EA	7	6	2	2
add reg32, mem32	0110 0110 0000 0011 [mod-reg-r/m]	-	-	-	6	2	2
add mem8, reg8	0000 0000 [mod-reg-r/m]	16+EA	16+EA	7	7	3	3
add mem16, reg16	0000 0001 [mod-reg-r/m]	24+EA	16+EA	7	7	3	3
add mem32, reg32	0110 0110 0000 0001 [mod-reg-r/m]	-	-	-	7	3	3
add reg8, imm8	1000 00x0 [11-000-r/m] [imm]	4	4	3	2	1	1
add reg16, imm16	1000 00s0 [11-000-r/m] [imm]	4	4	3	2	1	1
add reg32, imm32	0110 0110 1000 00s0 [11-000-r/m] [imm]	4	4	3	2	1	1
add mem8, imm8	1000 00x0 [mod-000-r/m] [imm]	17+EA	17+EA	7	7	3	3
add mem16, imm16	1000 00s1 [mod-000-r/m] [imm]	23+EA	17+EA	7	7	3	3
add mem32, imm32	0110 0110 1000 00s1 [mod-000-r/m] [imm]	-	-	-	7	3	3
add al, imm	0000 0100 [imm]	4	4	3	2	1	1
add ax, imm	0000 0101 [imm]	4	4	3	2	1	1
add eax, imm	0110 0110 0000 0101 [imm]	-	-	-	2	1	1
and reg8, reg8	0010 00x0 [11-reg-r/m]	3	‘3	2	2	1	1
and reg16, reg16	0010 00x1 [11-reg-r/m]	3	3	2	2	1	1
and reg32, reg32	0110 0110 0010 00x1 [11-reg-/rm]	3	3	2	2	1	1
and reg8, mem8	0010 0010 [mod-reg-r/m]	9+EA	9+EA	7	6	2	2

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
and reg16, mem16	0010 0011 [mod-reg-r/m]	13+EA	9+EA	7	6	2	2
and reg32, mem32	0110 0110 0010 0011 [mod-reg-r/m]	-	-	-	6	2	2
and mem8, reg8	0010 0000 [mod-reg-r/m]	16+EA	16+EA	7	7	3	3
and mem16, reg16	0010 0001 [mod-reg-r/m]	24+EA	16+EA	7	7	3	3
and mem32, reg32	0110 0110 0010 0001 [mod-reg-r/m]	-	-	-	7	3	3
and reg8, imm8	1000 00x0 [11-100-r/m] [imm]	4	4	3	2	1	1
and reg16, imm16	1000 00s1 [11-100-r/m] [imm]	4	4	3	2	1	1
and reg32, imm32	0110 0110 1000 00s1 [11-100-r/m] [imm]	4	4	3	2	1	1
and mem8, imm8	1000 00x0 [mod-100-r/m] [imm]	17+EA	17+EA	7	7	3	3
and mem16, imm16	1000 00s1 [mod-100-r/m] [imm]	23+EA	17+EA	7	7	3	3
and mem32, imm32	0110 0110 1000 00s1 [mod-100-r/m] [imm]	-	-	-	7	3	3
and al, imm	0010 0100 [imm]	4	4	3	2	1	1
and ax, imm	0010 0101 [imm]	4	4	3	2	1	1
and eax, imm	0110 0110 0010 0101 [imm]	-	-	-	2	1	1
bound reg16, mem32	0110 0010 [mod-reg-r/m]			13 (values within range)	10	7	8
bound reg32, mem64	0110 0110 0110 0010 [mod-reg-r/m]				10 (values within range)	7	8
bsf reg16, reg16	0000 1111 1011 1100 [11-reg-r/m]				10+3*n n= first set bit.	6-42	6-34

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
bsf reg32, reg32	0110 0110 0000 1111 1011 1100 [11-reg-r/m]				10+3*n n= first set bit.	6-42	6-42
bsf reg16, mem16	0000 1111 1011 1100 [mod-reg-r/m]				10+3*n n= first set bit.	7-43	6-35
bsf reg32, mem32	0110 0110 0000 1111 1011 1100 [mod-reg-r/m]				10+3*n n= first set bit.	7-43	6-43
bsr reg16, reg16	0000 1111 1011 1101 [11-reg-r/m]				10+3*n n= first set bit.	7-100	7-39
bsr reg32, reg32	0110 0110 0000 1111 1011 1101 [11-reg-r/m]				10+3*n n= first set bit.	8-100	7-71
bsr reg16, mem16	0000 1111 1011 1101 [mod-reg-r/m]				10+3*n n= first set bit.	7-101	7-40
bsr reg32, mem32	0110 0110 0000 1111 1011 1101 [mod-reg-r/m]				10+3*n n= first set bit.	8-101	7-72
bswap reg32	0000 1111 11001rrr					1	1
bt reg16, reg16	0000 1111 1010 0011 [11-reg-r/m]				3	3	4
bt reg32, reg32	0110 0110 0000 1111 1010 0011 [11-reg-r/m]				3	3	4
bt mem16, reg16	0000 1111 1010 0011 [mod-reg-r/m]				12	8	9
bt mem32, reg32	0110 0110 0000 1111 1010 0011 [mod-reg-r/m]				12	8	9
bt reg16, imm	0000 1111 1011 1010 [11-100-r/m] [imm8]				3	3	4
bt reg32, imm	0110 0110 0000 1111 1011 1010 [11-100-r/m] [imm8]				3	3	4
bt mem16, imm	0000 1111 1011 1010 [mod-100-r/m]				6	3	4

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
bt mem32, imm	0110 0110 0000 1111 1011 1010 [mod-100-r/m]				6	3	4
btc reg16, reg16	0000 1111 1011 1011 [11-reg-r/m]				6	6	7
btc reg32, reg32	0110 0110 0000 1111 1011 1011 [11-reg-r/m]				6	6	7
btc mem16, reg16	0000 1111 1011 1011 [mod-reg-r/m]				13	13	13
btc mem32, reg32	0110 0110 0000 1111 1011 1011 [mod-reg-r/m]				13	13	13
btc reg16, imm	0000 1111 1011 1010 [11-111-r/m] [imm8]				6	6	7
btc reg32, imm	0110 0110 0000 1111 1011 1010 [11-111-r/m] [imm8]				6	6	7
btc mem16, imm	0000 1111 1011 1010 [mod-111-r/m] [imm8]				8	8	8
btc mem32, imm	0110 0110 0000 1111 1011 1010 [mod-111-r/m] [imm8]				8	8	8
btr reg16, reg16	0000 1111 1011 0011 [11-reg-r/m]				6	6	7
btr reg32, reg32	0110 0110 0000 1111 1011 0011 [11-reg-r/m]				6	6	7
btr mem16, reg16	0000 1111 1011 0011 [mod-reg-r/m]				13	13	13
btr mem32, reg32	0110 0110 0000 1111 1011 0011 [mod-reg-r/m]				13	13	13

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
btr reg16, imm	0000 1111 1011 1010 [11-110-r/m] [imm8]				6	6	7
btr reg32, imm	0110 0110 0000 1111 1011 1010 [11-110-r/m] [imm8]				6	6	7
btr mem16, imm	0000 1111 1011 1010 [mod-110-r/m] [imm8]				8	8	8
btr mem32, imm	0110 0110 0000 1111 1011 1010 [mod-110-r/m] [imm8]				8	8	8
bts reg16, reg16	0000 1111 1010 1011 [11-reg-r/m]				6	6	7
bts reg32, reg32	0110 0110 0000 1111 1010 1011 [11-reg-r/m]				6	6	7
bts mem16, reg16	0000 1111 1010 1011 [mod-reg-r/m]				13	13	13
bts mem32, reg32	0110 0110 0000 1111 1010 1011 [mod-reg-r/m]				13	13	13
bts reg16, imm	0000 1111 1011 1010 [11-101-r/m] [imm8]				6	6	7
bts reg32, imm	0110 0110 0000 1111 1011 1010 [11-101-r/m] [imm8]				6	6	7
bts mem16, imm	0000 1111 1011 1010 [mod-101-r/m] [imm8]				8	8	8
bts mem32, imm	0110 0110 0000 1111 1011 1010 [mod-101-r/m] [imm8]				8	8	8
call near	1110 1000 [disp16]	23	19	7-10	7-10	3	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
call far	1001 1010 [offset] [segment]	36	28	13-16	17-20	18	4
call reg16	1111 1111 [11-010-r/m]	20	16	7-10	7-10	5	2
call mem16	1111 1111 [mod-010-r/m]	29+EA	21+EA	11-14	10-13	5	2
call mem32	1111 1111 [mod-011-r/m]	53+EA	37+EA	16-19	22-25	17	5
cbw	1001 1000	2	2	2	3	3	3
cdq	0110 0110 1001 1001				2	2	2
clc	1111 1000	2	2	2	2	2	2
cld	1111 1100	2	2	2	2	2	2
cli	1111 1010	2	2	3		5	7
cmc	1111 0101	2	2	2	2	2	2
cmp reg8, reg8	0011 10x0 [11-reg-r/m]	3	‘3	2	2	1	1
cmp reg16, reg16	0011 10x1 [11-reg-r/m]	3	3	2	2	1	1
cmp reg32, reg32	0110 0110 0011 10x1 [11-reg-/rm]	3	3	2	2	1	1
cmp reg8, mem8	0011 1010 [mod-reg-r/m]	9+EA	9+EA	7	6	2	2
cmp reg16, mem16	0011 1011 [mod-reg-r/m]	13+EA	9+EA	7	6	2	2
cmp reg32, mem32	0110 0110 0011 1011 [mod-reg-r/m]	-	-	-	6	2	2
cmp mem8, reg8	0011 1000 [mod-reg-r/m]	9+EA	9+EA	7	6	2	2
cmp mem16, reg16	0011 1001 [mod-reg-r/m]	13+EA	9+EA	7	6	2	2
cmp mem32, reg32	0110 0110 0011 1001 [mod-reg-r/m]	-	-	-	6	2	2
cmp reg8, imm8	1000 00x0 [11-111-r/m] [imm]	4	4	3	2	1	1
cmp reg16, imm16	1000 00s0 [11-111-r/m] [imm]	4	4	3	2	1	1
cmp reg32, imm32	0110 0110 1000 00s0 [11-111-r/m] [imm]	4	4	3	2	1	1
cmp mem8, imm8	1000 00x0 [mod-111-r/m] [imm]	10+EA	10+EA	6	5	2	2

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
cmp mem16, imm16	1000 00s1 [mod-111-r/m] [imm]	14+EA	10+EA	6	5	2	2
cmp mem32, imm32	0110 0110 1000 00s1 [mod-111-r/m] [imm]	-	-	-	5	2	2
cmp al, imm	0011 1100 [imm]	4	4	3	2	1	1
cmp ax, imm	0011 1101 [imm]	4	4	3	2	1	1
cmp eax, imm	0110 0110 0011 1101 [imm]	-	-	-	2	1	1
cmpsb	1010 0110	30	22	8	10	8	5
cmpsw	1010 0111	30	22	8	10	8	5
cmpsd	0110 0110 1010 0111	-	-	-	10	8	5
repe cmpsb	1111 0011 1010 0110	9+17*cx cx = # of repetitions	9+17*cx	5+9*cx	5+9*cx	7+7*cx 5 if cx=0	9+4*cx 7 if cx=0
repne cmpsb	1111 0010 1010 0110	9+17*cx	9+17*cx	5+9*cx	5+9*cx	7+7*cx 5 if cx=0	9+4*cx 7 if cx=0
repe cmpsw	1111 0011 1010 0111	9+25*cx	9+17*cx	5+9*cx	5+9*cx	7+7*cx 5 if cx=0	9+4*cx 7 if cx=0
repne cmpsw	1111 0010 1010 0111	9+25*cx	9+17*cx	5+9*cx	5+9*cx	7+7*cx 5 if cx=0	9+4*cx 7 if cx=0
repe cmpsd	0110 0110 1111 0011 1010 0111	-	-	-	5+9*cx	7+7*cx 5 if cx=0	9+4*cx 7 if cx=0
repne cmpsd	0110 0110 1111 0010 1010 0111	-	-	-	5+9*cx	7+7*cx 5 if cx=0	9+4*cx 7 if cx=0
cmpxchg reg8, reg8	0000 1111 1011 0000 [11-reg-r/m] Note: r/m is first register operand.	-	-	-	-	6	6
cmpxchg reg16, reg16	0000 1111 1011 0001 [11-reg-r/m]	-	-	-	-	6	6
cmpxchg reg32, reg32	0110 0110 0000 1111 1011 0001 [11-reg-r/m]	-	-	-	-	6	6
cmpxchg mem8, reg8	0000 1111 1011 0000 [mod-reg-r/m]	-	-	-	-	7 if equal, 10 if not equal	6

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
cmpxchg mem16, reg16	0000 1111 1011 0001 [mod-reg-r/m]	-	-	-	-	7 if equal, 10 if not equal	6
cmpxchg mem32, reg32	0110 0110 0000 1111 1011 0001 [mod-reg-r/m]	-	-	-	-	7 if equal, 10 if not equal	6
cmpxchg8b mem64	0000 1111 1100 0111 [mod-001-r/m]	-	-	-	-	-	10
cpuid	0000 1111 1010 0010	-	-	-	-	-	14
cwd	1001 1001	5	5	2	2	3	2
cwde	0110 0110 1001 1000				3	3	3
daa	0010 0111	4	4	3	4	2	3
das	0010 1111	4	4	3	4	2	3
dec reg8	1111 1110 [11-001-r/m]	3	3	2	2	1	1
dec reg16	0100 1rrr	3	3	2	2	1	1
dec reg16 (alternate encoding)	1111 1111 [11-001-r/m]	3	3	2	2	1	1
dec reg32	0110 0110 0100 1rrr	3	3	2	2	1	1
dec reg32 (alternate encoding)	0110 0110 1111 1111 [11-001-r/m]	3	3	2	2	1	1
dec mem8	1111 1110 [mod-001-r/m]	15+EA	15+EA	7	6	3	3
dec mem16	1111 1111 [mod-001-r/m]	23+EA	15+EA	7	6	3	3
dec mem32	0110 0110 1111 1111 [mod-001-r/m]	-	-	-	6	3	3
div reg8	1111 0110 [11-110-r/m]	80-90	80-90	14	14	16	17
div reg16	1111 0111 [11-110-r/m]	144-162	144-162	22	22	24	25
div reg32	0110 0110 1111 0111 [11-110-r/m]	-	-	-	38	40	41
div mem8	1111 0110 [mod-110-r/m]	(86-96) + EA	(86-96) + EA	17	17	16	17
div mem16	1111 0111 [mod-110-r/m]	(158-176) + EA	(150-168) + EA	25	25	24	25
div mem32	0110 0110 1111 0111 [mod-110-r/m]	-	-	-	41	40	41

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
enter local, 0	1100 1000 [locals-imm16] 0000 0000			11	10	14	11
enter local, 1	1100 1000 [locals-imm16] 0000 0001			15	12	17	15
enter local, lex	1100 1000 [locals:imm16] [lex:imm8]			12 + 4 * (lex-1)	15 + 4 * (lex-1)	17 + 3*lex	15 + 2*lex
hlt	1111 0100	2+[d]	2+	2+	5+	4+	12+
idiv reg8	1111 0110 [11-111-r/m]	101-112	101-112	17	19	19	22
idiv reg16	1111 0111 [11-111-r/m]	165-184	165-184	25	27	27	30
idiv reg32	0110 0110 1111 0111 [11-111-r/m]	-	-	-	43	43	46
idiv mem8	1111 0110 [mod-111-r/m]	(107-118) + EA	(107-118) + EA	20	22	20	30
idiv mem16	1111 0111 [mod-111-r/m] [disp]	(175-194) + EA	(171-190) + EA	28	30	28	30
idiv mem32	0110 0110 1111 0111 [mod-111-r/m]	-	-	-	46	44	46
imul reg8	1111 0110 [11-101-r/m]	80-98	80-98	13	9-14	13-18	11
imul reg16	1111 0111 [11-101-r/m]	128-154	128-154	21	9-22	13-26	11
imul reg32	0110 0110 1111 0111 [11-101-r/m]	-	-	-	9-38	13-42	11
imul mem8	1111 0110 [mod-101-r/m]	(86-104) + EA	(107-118) + EA	16	12-17	13-18	11
imul mem16	1111 0111 [mod-101-r/m]	(134-164) + EA	(134-160) + EA	24	15-25	13-26	11
imul mem32	0110 0110 1111 0111 [mod-101-r/m]	-	-	-	12-41	13-42	11
imul reg16, reg16, imm8 imul reg16, imm8 (Second form assumes reg and r/m are the same, instruction sign extends eight bit immediate operand to 16 bits)	0110 1011 [11-reg-r/m] [imm8] (1st reg operand is specified by reg field, 2nd reg operand is specified by r/m field)	-	-	21	13-26	13-26	10
imul reg16, reg16, imm imul reg16, imm	0110 1001 [11-reg-r/m] [imm16]	-	-	21	9-22	13-26	10

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
imul reg32, reg32, imm8 imul reg32, imm8	0110 0110 0110 1011 [11-reg-r/m] [imm8]	-	-		13-42	13-42	10
imul reg32, reg32, imm imul reg32, imm	0110 0110 0110 1001 [11-reg-r/m] [imm32]	-	-	-	9-38	13-42	10
imul reg16,mem16,imm8	0110 1011 [11-reg-r/m] [imm8]	-	-	24	14-27	13-26	10
imul reg16,mem16,imm	0110 1001 [11-reg-r/m] [imm16]	-	-	24	12-25	13-26	10
imul reg32, mem32, imm8	0110 0110 0110 1011 [11-reg-r/m] [imm8]	-	-	-	14-43	13-42	10
imul reg32, mem32, imm	0110 0110 0110 1001 [11-reg-r/m] [imm32]	-	-	-	12-41	13-42	10
imul reg16, reg16	0000 1111 1010 1111 [11-reg-r/m] (reg is dest operand)	-	-	-	12-25	13-26	10
imul reg32, reg32	0110 0110 0000 1111 1010 1111 [11-reg-r/m] (reg is dest operand)	-	-	-	12-41	12-42	10
imul reg16, mem16	0000 1111 1010 1111 [mod-reg-r/m]	-	-	-	15-28	13-26	10
imul reg32, mem32	0110 0110 0000 1111 1010 1111 [mod-reg-r/m]	-	-	-	14-44	13-42	10
in al, port	1110 0100 [port8]	10	10	5	12	14	7
in ax, port	1110 0101 [port8]	14	10	5	12	14	7
in eax, port	0110 0110 1110 0101 [port8]	-	-	-	12	14	7
in al, dx	1110 1100	8	8	5	13	14	7
in ax, dx	1110 1101	12	8	5	13	14	7
in eax, dx	0110 0110 1110 1101	12	8	5	13	14	7

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
inc reg8	1111 1110 [11-000-r/m]	3	2	2	2	1	1
inc reg16	0100 0rrr	3	3	2	2	1	1
inc reg16 (alternate encoding)	1111 1111 [11-000-r/m]	3	3	2	2	1	1
inc reg32	0110 0110 0100 0rrr	-	-	-	2	1	1
inc reg32 (alternate encoding)	0110 0110 1111 1111 [11-000-r/m]	-	-	-	2	1	1
inc mem8	1111 1110 [mod-000-r/m]	15+EA	15+EA	7	6	3	3
inc mem16	1111 1110 [mod-000-r/m] [disp]	23+EA	15+EA	7	6	3	3
inc mem32	0110 0110 1111 1110 [mod-000-r/m]	-	-	-	6	3	3
insb	1010 1010	-	-	5	15	17	9
insw	1010 1011	-	-	5	15	17	9
insd	0110 0110 1010 1011	-	-	-	15	17	9
rep insb	1111 0010 1010 1010	-	-	5 + 4*cx	14 + 6*cx	16+8*cx	11 + 3*cx
rep insw	1111 0010 1010 1011	-	-	5 + 4*cx	14 + 6*cx	16+8*cx	11 + 3*cx
rep insd	0110 0110 1111 0010 1010 1011	-	-	-	14 + 6*cx	16+8*cx	11 + 3*cx
int nn	1100 1101 [imm8]	71	51	23-26	37	30	16
int 03	1100 1100	72	52	23-26	33	26	13
into	1100 1110	73 (if ovr) 4 (no ovr)	53 4	24-27 3	35 3	28 3	13 3
iret	1100 1111	44	32	17-20	22	15	8
iretd	0110 0110 1100 1111				22	15	10
ja short	0111 0111 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
ja near	0000 1111 1000 0111 [disp16]	-	-	-	7-10 3	3 1	1
jae short	0111 0011 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jae near	0000 1111 1000 0011 [disp16]	-	-	-	7-10 3	3 1	1
jb short	0111 0010 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
jb near	0000 1111 1000 0010 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jbe short	0111 0110 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jbe near	0000 1111 1000 0110 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jc short	0111 0010 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jc near	0000 1111 1000 0010 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
je short	0111 0100 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
je near	0000 1111 1000 0100 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jg short	0111 1111 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jg near	0000 1111 1000 1111 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jge short	0111 1101 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jge near	0000 1111 1000 1101 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jl short	0111 1100 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jl near	0000 1111 1000 1100 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jle short	0111 1110 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jle near	0000 1111 1000 1110 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jna short	0111 0110 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jna near	0000 1111 1000 0110 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jnae short	0111 0010 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnae near	0000 1111 1000 0010 [disp16]	- 	- 	- 	7-10 3 	3 1 	1
jnb short	0111 0011 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
jnb near	0000 1111 1000 0011 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnbe short	0111 0111 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnbe near	0000 1111 1000 0111 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnc short	0111 0011 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnc near	0000 1111 1000 0011 [disp16]	- 	- 	- 	7-10 3	3 1	1
jne short	0111 0101 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jne near	0000 1111 1000 0101 [disp16]	- 	- 	- 	7-10 3	3 1	1
jng short	0111 1110 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jng near	0000 1111 1000 1110 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnge short	0111 1100 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnge near	0000 1111 1000 1100 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnl short	0111 1101 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnl near	0000 1111 1000 1101 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnle short	0111 1111 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnle near	0000 1111 1000 1111 [disp16]	- 	- 	- 	7-10 3	3 1	1
jno short	0111 0001 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jno near	0000 1111 1000 0001 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnp short	0111 1011 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnp near	0000 1111 1000 1011 [disp16]	- 	- 	- 	7-10 3	3 1	1
jns short	0111 1001 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
jns near	0000 1111 1000 1001 [disp16]	- 	- 	- 	7-10 3	3 1	1
jnz short	0111 0101 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jnz near	0000 1111 1000 0101 [disp16]	- 	- 	- 	7-10 3	3 1	1
jo short	0111 0000 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jo near	0000 1111 1000 0000 [disp16]	- 	- 	- 	7-10 3	3 1	1
jp short	0111 1010 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jp near	0000 1111 1000 1010 [disp16]	- 	- 	- 	7-10 3	3 1	1
jpe short	0111 1010 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jpe near	0000 1111 1000 1010 [disp16]	- 	- 	- 	7-10 3	3 1	1
jpo short	0111 1011 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jpo near	0000 1111 1000 1011 [disp16]	- 	- 	- 	7-10 3	3 1	1
js short	0111 1000 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
js near	0000 1111 1000 1000 [disp16]	- 	- 	- 	7-10 3	3 1	1
jz short	0111 0100 [disp8]	16 4 (not taken)	16 4	7-10 3	7-10 3	3 1	1
jz near	0000 1111 1000 0100 [disp16]	- 	- 	- 	7-10 3	3 1	1
jcxz short	1110 0011 [disp8]	18 6 (not taken)	18 6	8-11 4	9-12 5	8 5	6 5
jecxz short	0110 0110 1110 0011 [disp8]				9-12 5	8 5	6 5
jmp short	1110 1011 [disp8]	15	15	7-10	7-10	3	1
jmp near	1110 1001 [disp16]	15	15	7-10	7-10	3	1
jmp reg16	1111 1111 [11-100-r/m]	11	11	7-10	7-10	5	2

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
jmp mem16	1111 1111 [mod-100-r/m]	18+EA	18+EA	11-14	10-13	5	2
jmp far	1110 1010 [offset16] [segment16]	15	15	11-14	12-15	17	3
jmp mem32	1111 1111 [mod-101-r/m]	24+EA	24+EA	15-18	43-46	13	2
lahf	1001 1111	4	4	2	2	3	2
lds reg, mem32	1100 0101 [mod-reg-r/m]	24+EA	16+EA	7	7	6	4
lea reg, mem	1000 1101 [mod-101-r/m]	2+EA	2+EA	3	2	1	1
leave	1100 1001	-	-	5	4	5	3
les reg, mem32	1100 0100 [mod-reg-r/m]	24+EA	16+EA	7	7	6	4
lfs reg, mem32	0000 1111 1011 0100 [mod-reg-r/m]	-	-	-	7	6	4
lgs reg, mem32	0000 1111 1011 0101 [mod-reg-r/m]	-	-	-	7	6	4
lodsb	1010 1100	12	12	5	5	5	2
lodsw	1010 1101	16	12	5	5	5	2
loadsd	0110 0110 1010 1101	-	-	-	5	5	2
loop short	1110 0010 [disp8]	17 5 (not taken)	17 5	8-11 4	11-14	7 6	5
loope short loopz short	1110 0001 [disp8]	18 6 (not taken)	18 6	8-11 4	11-14	9 6	7
loopne short loopnz short	1110 0000 [disp8]	19 5(not taken)	19 5	8-11 4	11-14	9 6	7
lss reg, mem32	0000 1111 1011 0010 [mod-reg-r/m]	-	-	-	7	6	4
mov reg8, reg8	1000 1000 [11-reg-r/m] (r/m specifies destination reg)	2	2	2	2	1	1
mov reg8, reg8 (alternate encoding)	1000 1010 [11-reg-r/m] (reg specifies destination reg)	2	2	2	2	1	1
mov reg16, reg16	1000 1001 [11-reg-r/m] (r/m specifies destination reg)	2	2	2	2	1	1
mov reg16, reg16 (alternate encoding)	1000 1011 [11-reg-r/m] (reg specifies destination reg)	2	2	2	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
mov reg32, reg32	0110 0110 1000 1001 [11-reg-r/m] (r/m specifies destination reg)	-	-	-	2	1	1
mov reg32, reg32 (alternate encoding)	0110 0110 1000 1011 [11-reg-r/m] (reg specifies destination reg)	-	-	-	2	1	1
mov mem, reg8	1000 1000 [mod-reg-r/m]	9+EA	9+EA	3	2	1	1
mov reg8, mem	1000 1010 [mod-reg-r/m]	8+EA	8+EA	5	4	1	1
mov mem, reg16	1000 1001 [mod-reg-r/m]	13+EA	9+EA	3	2	1	1
mov reg16, mem	1000 1011 [mod-reg-r/m]	12+EA	8+EA	5	4	1	1
mov mem, reg32	0110 0110 1000 1001 [mod-reg-r/m]	-	-	-	2	1	1
mov reg16, mem	0110 0110 1000 1011 [mod-reg-r/m]	-	-	-	4	1	1
mov reg8, imm	1011 0rrr [imm8]	4	4	2	2	1	1
mov reg8, imm (alternate encoding)	1100 0110 [11-000-r/m] [imm8]	10	10	2	2	1	1
mov reg16, imm	1011 1rrr [imm16]	4	4	2	2	1	1
mov reg16, imm (alternate encoding)	1100 0111 [11-000-r/m] [imm16]	10	10	2	2	1	1
mov reg32, imm	0110 0110 1011 1rrr [imm32]	-	-	-	2	1	1
mov reg32, imm (alternate encoding)	0110 0110 1100 0111 [11-000-r/m] [imm32]	-	-	-	2	1	1
mov mem8, imm	1100 0110 [mod-000-r/m] [imm8]	10+EA	10+EA	3	2	1	1
mov mem16, imm	1100 0111 [mod-000-r/m] [imm16]	14+EA	10+EA	3	2	1	1
mov mem32, imm	1100 0111 [mod-000-r/m] [imm32]	-	-	-	2	1	1
mov al, disp	1010 0000 [disp]	10	10	5	4	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
mov ax, disp	1010 0001 [disp]	14	10	5	4	1	1
mov eax, disp	0110 0110 1010 0001 [disp]	-	-	-	4	1	1
mov disp, al	1010 0010 [disp]	10	10	3	2	1	1
mov disp, ax	1010 0011 [disp]	14	10	3	2	1	1
mov disp, eax	0110 0110 1010 0011 [disp]	-	-	-	2	1	1
mov segreg, reg16	1000 1110 [11-sreg-r/m]	2	2	2	2	3	2-3
mov segreg, mem	1000 1110 [mod-reg-r/m]	12+EA	8+EA	5	5	3	2-3
mov reg16, segreg	1000 1100 [11-sreg-r/m]	2	2	2	2	3	1
mov mem, segreg	1000 1100 [mod-reg-r/m]	13+EA	9+EA	3	2	3	1
movsb	1010 0100	18	18	5	8	7	4
movsw	1010 0101	26	18	5	8	7	4
movsd	0110 0110 1010 0101	-	-	-	8	7	4
rep movsb	1111 0010 1010 0100	9 + 17 * cx	9 + 17*cx	5 + 4*cx	8 + 4*cx	12 + 3*cx 5 if cx=0 13 if cx=1	4 + 3*cx
rep movsw	1111 0010 1010 0101	9 + 25 * cx	9 + 17*cx	5 + 4*cx	8 + 4*cx	12 + 3*cx 5 if cx=0 13 if cx=1	4 + 3*cx
rep movsd	0110 0110 1111 0010 1010 0101	-	-	-	8 + 4*cx	12 + 3*cx 5 if cx=0 13 if cx=1	4 + 3*cx
movsx reg16, reg8	0000 1111 1011 1110 [11-reg-r/m] (dest is reg operand)				3	3	3
movsx reg32, reg8	0110 0110 0000 1111 1011 1110 [11-reg-r/m]				3	3	3
movsx reg32, reg16	0110 0110 0000 1111 1011 1111 [11-reg-r/m]				3	3	3
movsx reg16, mem8	0000 1111 1011 1110 [mod-reg-r/m]				6	3	3

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
movsx reg32, mem8	0110 0110 0000 1111 1011 1110 [mod-reg-r/m]				6	3	3
movsx reg32, mem16	0110 0110 0000 1111 1011 1111 [mod-reg-r/m]				6	3	3
movzx reg16, reg8	0000 1111 1011 0110 [11-reg-r/m] (dest is reg operand)				3	3	3
movzx reg32, reg8	0110 0110 0000 1111 1011 0110 [11-reg-r/m]				3	3	3
movzx reg32, reg16	0110 0110 0000 1111 1011 0111 [11-reg-r/m]				3	3	3
movzx reg16, mem8	0000 1111 1011 0110 [mod-reg-r/m]				6	3	3
movzx reg32, mem8	0110 0110 0000 1111 1011 0110 [mod-reg-r/m]				6	3	3
movzx reg32, mem16	0110 0110 0000 1111 1011 0111 [mod-reg-r/m]				6	3	3
mul reg8	1111 0110 [11-100-r/m]	70-77	70-77	13	9-14	13-18	11
mul reg16	1111 0111 [11-100-r/m]	118-133	118-133	21	9-22	13-26	11
mul reg32	0110 0110 1111 0111 [11-100-r/m]	-	-	-	9-38	13-42	10
mul mem8	1111 0110 [mod-100-r/m]	(76-83) + EA	(76-83) + EA	16	12-17	13-18	11
mul mem16	1111 0111 [mod-100-r/m]	(124-139) + EA	(124-139) + EA	24	12-25	13-26	11
mul mem32	0110 0110 1111 0111 [mod-100-r/m]	-	-	-	12-41	13-42	10
neg reg8	1111 0110 [11-011-r/m]	3	3	2	2	1	1
neg reg16	1111 0111 [11-011-r/m]	3	3	2	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
neg reg32	0110 0110 1111 0111 [11-011-r/m]	3	3	2	2	1	1
neg mem8	1111 0110 [mod-011-r/m]	16+EA	16+EA	7	6	3	3
neg mem16	1111 0111 [mod-011-r/m]	24+EA	16+EA	7	6	3	3
neg mem32	0110 0110 1111 0111 [mod-011-r/m]	-	-	-	6	3	3
nop (same as xchg ax, ax)	1001 0000	3	3	3	3	1	1
not reg8	1111 0110 [11-010-r/m]	3	3	2	2	1	1
not reg16	1111 0111 [11-010-r/m]	3	3	2	2	1	1
not reg32	0110 0110 1111 0111 [11-010-r/m]	3	3	2	2	1	1
not mem8	1111 0110 [mod-010-r/m]	16+EA	16+EA	7	6	3	3
not mem16	1111 0111 [mod-010-r/m]	24+EA	16+EA	7	6	3	3
not mem32	0110 0110 1111 0111 [mod-010-r/m]	-	-	-	6	3	3
or reg8, reg8	0000 10x0 [11-reg-r/m]	3	‘3	2	2	1	1
or reg16, reg16	0000 10x1 [11-reg-r/m]	3	3	2	2	1	1
or reg32, reg32	0110 0110 0000 10x1 [11-reg-r/m]	3	3	2	2	1	1
or reg8, mem8	0000 1010 [mod-reg-r/m]	9+EA	9+EA	7	6	2	2
or reg16, mem16	0000 1011 [mod-reg-r/m]	13+EA	9+EA	7	6	2	2
or reg32, mem32	0110 0110 0000 1011 [mod-reg-r/m]	-	-	-	6	2	2
or mem8, reg8	0000 1000 [mod-reg-r/m]	16+EA	16+EA	7	7	3	3
or mem16, reg16	0000 1001 [mod-reg-r/m]	24+EA	16+EA	7	7	3	3
or mem32, reg32	0110 0110 0000 1001 [mod-reg-r/m]	-	-	-	7	3	3
or reg8, imm8	1000 00x0 [11-001-r/m] [imm]	4	4	3	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
or reg16, imm16	1000 00s0 [11-001-r/m] [imm]	4	4	3	2	1	1
or reg32, imm32	0110 0110 1000 00s0 [11-001-r/m] [imm]	4	4	3	2	1	1
or mem8, imm8	1000 00x0 [mod-001-r/m] [imm]	17+EA	17+EA	7	7	3	3
or mem16, imm16	1000 00s1 [mod-001-r/m] [imm]	25+EA	17+EA	7	7	3	3
or mem32, imm32	0110 0110 1000 00s1 [mod-001-r/m] [imm]	-	-	-	7	3	3
or al, imm	0000 1100 [imm]	4	4	3	2	1	1
or ax, imm	0000 10101 [imm]	4	4	3	2	1	1
or eax, imm	0110 0110 0000 1101 [imm]	-	-	-	2	1	1
out port, al	1110 0110 [port8]	14	10	3	10	16	12
out port, ax	1110 0111 [port8]	14	10	3	10	16	12
out port, eax	0110 0110 1110 0111 [port8]	-	-	-	10	16	12
out dx, al	1110 1110	8	8	3	11	16	12
out dx, ax	1110 1111	12	8	3	11	16	12
out dx, eax	0110 0110 1110 1111	-	-	-	11	16	12
outsb	1010 1010	-	-	5	14	17	13
outsw	1010 1011	-	-	5	14	17	13
outsd	0110 0110 1010 1011	-	-	-	14	17	13
rep outsb	1111 0010 1010 1010	-	-	5 + 4*cx	12 + 5*cx	17+5*cx	13 + 4*cx
rep outsw	1111 0010 1010 1011	-	-	5 + 4*cx	12 + 5*cx	17+5*cx	13 + 4*cx
rep outsd	0110 0110 1111 0010 1010 1011	-	-	-	12 + 5*cx	17+5*cx	13 + 4*cx
pop reg16	0101 1rrr	12	8	5	4	1	1
pop reg16 (alternate encoding)	1000 1111 [11-000-r/m]	12	8	5	4	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
pop reg32	0110 0110 0101 1rrr	-	-	-	4	1	1
pop reg32 (alternate encoding)	0110 0110 1000 1111 [11-000-r/m]	-	-	-	5	4	3
pop mem16	1000 1111 [mod-000-r/m]	25+EA	17+EA	5	5	6	3
pop mem32	1000 1111 [mod-000-r/m]	-	-	-	5	6	3
pop es	0000 0111	12	8	5	7	3	3
pop ss	0001 0111	12	8	5	7	3	3
pop ds	0001 1111	12	8	5	7	3	3
pop fs	0000 1111 1010 0001	-	-	-	7	3	3
pop gs	0000 1111 1010 1001	-	-	-	7	3	3
popa	0110 0001	-	-	19	24	9	5
popad	0110 0110 0110 0001	-	-	-	24	9	5
popf	1001 1101	12	8	5	5	9	6
popfd	0110 0110 1001 1101	-	-	-	5	9	6
push reg16	0101 0rrr	15	11	3	2	1	1
push reg16 (alternate encoding)	1111 1111 [11-110-r/m]	15	11	3	2	1	1
push reg32	0110 0110 0101 0rrr	-	-	-	2	1	1
push reg32 (alternate encoding)	0110 0110 1111 1111 [11-110-r/m]	-	-	-	2	1	1
push mem16	1111 1111 [mod-110-r/m]	24+EA	16+EA	5	5	4	2
push mem32	1111 1111 [mod-110-r/m]	-	-	-	5	4	2
push cs	0000 1110	14	10	3	2	3	1
push ds	0001 1110	14	10	3	2	3	1
push es	0000 0110	14	10	3	2	3	1
push ss	0001 0110	14	10	3	2	3	1
push fs	0000 1111 1010 0000	-	-	-	2	3	1
push gs	0000 1111 1010 1000	-	-	-	2	3	1
push imm8->16	0110 1000 [imm8] (sign extends value to 16 bits)	-	-	3	2	1	1
push imm16	0110 1010 [imm16]	-	-	3	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
push imm32	0110 0110 0110 1010 [imm32]	-	-	-	2	1	1
pusha	0110 0000	-	-	17	18	11	5
pushad	0110 0110 0110 0000	-	-	-	18	11	5
pushf	1001 1100	14	10	3	4	4	4
pushfd	0110 0110 1001 1100	-	-	-	4	4	4
rcl reg8, 1	1101 0000 [11-010-r/m]	2	2	2	9	3	1
rcl reg16, 1	1101 0001 [11-010-r/m]	2	2	2	9	3	1
rcl reg32, 1	0110 0110 1101 0001 [11-010-r/m]	-	-	-	9	3	1
rcl mem8, 1	1101 0000 [mod-010-r/m]	15+EA	15+EA	7	10	4	3
rcl mem16, 1	1101 0001 [mod-010-r/m]	23+EA	15+EA	7	10	4	3
rcl mem32, 1	0110 0110 1101 0001 [mod-010-r/m]	-	-	-	10	4	3
rcl reg8, cl	1101 0010 [11-010-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	9	8-30	7-24
rcl reg16, cl	1101 0011 [11-010-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	9	8-30	7-24
rcl reg32, cl	0110 0110 1101 0011 [11-010-r/m]	-	-	-	9	8-30	7-24
rcl mem8, cl	1101 0010 [mod-010-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	10	9-31	9-26
rcl mem16, cl	1101 0011 [mod-010-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	10	9-31	9-26
rcl mem32, cl	0110 0110 1101 0011 [mod-010-r/m]	-	-	-	10	9-31	9-26
rcl reg8, imm8	1100 0000 [11-010-r/m] [imm8]	-	-	5+imm8	9	8-30	8-25
rcl reg16, imm8	1100 0001 [11-010-r/m] [imm8]	-	-	5+imm8	9	8-30	8-25
rcl reg32, imm8	0110 0110 1100 0001 [11-010-r/m] [imm8]	-	-	-	9	8-30	8-25
rcl mem8, imm8	1100 0000 [mod-010-r/m] [imm8]	-	-	8+imm8	10	9-31	10-27

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
rcl mem16, imm8	1100 0001 [mod-010-r/m] [imm8]	-	-	8+imm8	10	9-31	10-27
rcl mem32, imm8	0110 0110 1100 0001 [mod-010-r/m] [imm8]	-	-	-	10	9-31	10-27
rcr reg8, 1	1101 0000 [11-011-r/m]	2	2	2	9	3	1
rcr reg16, 1	1101 0001 [11-011-r/m]	2	2	2	9	3	1
rcr reg32, 1	0110 0110 1101 0001 [11-011-r/m]	-	-	-	9	3	1
rcr mem8, 1	1101 0000 [mod-011-r/m]	15+EA	15+EA	7	10	4	3
rcr mem16, 1	1101 0001 [mod-011-r/m]	23+EA	15+EA	7	10	4	3
rcr mem32, 1	0110 0110 1101 0001 [mod-011-r/m]	-	-	-	10	4	3
rcr reg8, cl	1101 0010 [11-011-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	9	8-30	7-24
rcr reg16, cl	1101 0011 [11-011-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	9	8-30	7-24
rcr reg32, cl	0110 0110 1101 0011 [11-011-r/m]	-	-	-	9	8-30	7-24
rcr mem8, cl	1101 0010 [mod-011-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	10	9-31	9-26
rcr mem16, cl	1101 0011 [mod-011-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	10	9-31	9-26
rcr mem32, cl	0110 0110 1101 0011 [mod-011-r/m]	-	-	-	10	9-31	9-26
rcr reg8, imm8	1100 0000 [11-011-r/m] [imm8]	-	-	5+imm8	9	8-30	8-25
rcr reg16, imm8	1100 0001 [11-011-r/m] [imm8]	-	-	5+imm8	9	8-30	8-25
rcr reg32, imm8	0110 0110 1100 0001 [11-011-r/m] [imm8]	-	-	-	9	8-30	8-25
rcr mem8, imm8	1100 0000 [mod-011-r/m] [imm8]	-	-	8+imm8	10	9-31	10-27
rcr mem16, imm8	1100 0001 [mod-011-r/m] [imm8]	-	-	8+imm8	10	9-31	10-27

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
rcr mem32, imm8	0110 0110 1100 0001 [mod-011-r/m] [imm8]	-	-	-	10	9-31	10-27
ret retn	1100 0011	20	16	11-14	10-13	5	2
ret imm16 retn imm16	1100 0010 [imm16]	24	20	11-14	10-13	5	3
ret retf	1100 1011	34	26	15-18	18-21	13	4
ret imm16 retf imm16	1100 1010 [imm16]	33	25	15-18	18-21	14	4
rol reg8, 1	1101 0000 [11-000-r/m]	2	2	2	3	3	1
rol reg16, 1	1101 0001 [11-000-r/m]	2	2	2	3	3	1
rol reg32, 1	0110 0110 1101 0001 [11-000-r/m]	-	-	-	3	3	1
rol mem8, 1	1101 0000 [mod-000-r/m]	15+EA	15+EA	7	7	4	3
rol mem16, 1	1101 0001 [mod-000-r/m]	23+EA	15+EA	7	7	4	3
rol mem32, 1	0110 0110 1101 0001 [mod-000-r/m]	-	-	-	7	4	3
rol reg8, cl	1101 0010 [11-000-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
rol reg16, cl	1101 0011 [11-000-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
rol reg32, cl	0110 0110 1101 0011 [11-000-r/m]	-	-	-	3	3	4
rol mem8, cl	1101 0010 [mod-000-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
rol mem16, cl	1101 0011 [mod-000-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
rol mem32, cl	0110 0110 1101 0011 [mod-000-r/m]	-	-	-	7	4	4
rol reg8, imm8	1100 0000 [11-000-r/m] [imm8]	-	-	5+imm8	3	2	1
rol reg16, imm8	1100 0001 [11-000-r/m] [imm8]	-	-	5+imm8	3	2	1
rol reg32, imm8	0110 0110 1100 0001 [11-000-r/m] [imm8]	-	-	-	3	2	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
rol mem8, imm8	1100 0000 [mod-000-r/m] [imm8]	-	-	8+imm8	7	4	3
rol mem16, imm8	1100 0001 [mod-000-r/m] [imm8]	-	-	8+imm8	7	4	3
rol mem32, imm8	0110 0110 1100 0001 [mod-000-r/m] [imm8]	-	-	-	7	4	3
ror reg8, 1	1101 0000 [11-001-r/m]	2	2	2	3	3	1
ror reg16, 1	1101 0001 [11-001-r/m]	2	2	2	3	3	1
ror reg32, 1	0110 0110 1101 0001 [11-001-r/m]	-	-	-	3	3	1
ror mem8, 1	1101 0000 [mod-001-r/m]	15+EA	15+EA	7	7	4	3
ror mem16, 1	1101 0001 [mod-001-r/m]	23+EA	15+EA	7	7	4	3
ror mem32, 1	0110 0110 1101 0001 [mod-001-r/m]	-	-	-	7	4	3
ror reg8, cl	1101 0010 [11-001-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
ror reg16, cl	1101 0011 [11-001-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
ror reg32, cl	0110 0110 1101 0011 [11-001-r/m]	-	-	-	3	3	4
ror mem8, cl	1101 0010 [mod-001-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
ror mem16, cl	1101 0011 [mod-001-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
ror mem32, cl	0110 0110 1101 0011 [mod-001-r/m]	-	-	-	7	4	4
ror reg8, imm8	1100 0000 [11-001-r/m] [imm8]	-	-	5+imm8	3	2	1
ror reg16, imm8	1100 0001 [11-001-r/m] [imm8]	-	-	5+imm8	3	2	1
ror reg32, imm8	0110 0110 1100 0001 [11-001-r/m] [imm8]	-	-	-	3	2	1
ror mem8, imm8	1100 0000 [mod-001-r/m] [imm8]	-	-	8+imm8	7	4	3

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
ror mem16, imm8	1100 0001 [mod-001-r/m] [imm8]	-	-	8+imm8	7	4	3
ror mem32, imm8	0110 0110 1100 0001 [mod-001-r/m] [imm8]	-	-	-	7	4	3
sahf	1001 1110	4	4	2	3	2	2
sal reg8, 1 (Same instruction as shl)	1101 0000 [11-100-r/m]	2	2	2	3	3	1
sal reg16, 1	1101 0001 [11-100-r/m]	2	2	2	3	3	1
sal reg32, 1	0110 0110 1101 0001 [11-100-r/m]	-	-	-	3	3	1
sal mem8, 1	1101 0000 [mod-100-r/m]	15+EA	15+EA	7	7	4	3
sal mem16, 1	1101 0001 [mod-100-r/m]	23+EA	15+EA	7	7	4	3
sal mem32, 1	0110 0110 1101 0001 [mod-100-r/m]	-	-	-	7	4	3
sal reg8, cl	1101 0010 [11-100-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
sal reg16, cl	1101 0011 [11-100-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
sal reg32, cl	0110 0110 1101 0011 [11-100-r/m]	-	-	-	3	3	4
sal mem8, cl	1101 0010 [mod-100-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
sal mem16, cl	1101 0011 [mod-100-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
sal mem32, cl	0110 0110 1101 0011 [mod-100-r/m]	-	-	-	7	4	4
sal reg8, imm8	1100 0000 [11-100-r/m] [imm8]	-	-	5+imm8	3	2	1
sal reg16, imm8	1100 0001 [11-100-r/m] [imm8]	-	-	5+imm8	3	2	1
sal reg32, imm8	0110 0110 1100 0001 [11-100-r/m] [imm8]	-	-	-	3	2	1
sal mem8, imm8	1100 0000 [mod-100-r/m] [imm8]	-	-	8+imm8	7	4	3

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
sal mem16, imm8	1100 0001 [mod-100-r/m] [imm8]	-	-	8+imm8	7	4	3
sal mem32, imm8	0110 0110 1100 0001 [mod-100-r/m] [imm8]	-	-	-	7	4	3
sar reg8, 1	1101 0000 [11-111-r/m]	2	2	2	3	3	1
sar reg16, 1	1101 0001 [11-111-r/m]	2	2	2	3	3	1
sar reg32, 1	0110 0110 1101 0001 [11-111-r/m]	-	-	-	3	3	1
sar mem8, 1	1101 0000 [mod-111-r/m]	15+EA	15+EA	7	7	4	3
sar mem16, 1	1101 0001 [mod-111-r/m]	23+EA	15+EA	7	7	4	3
sar mem32, 1	0110 0110 1101 0001 [mod-111-r/m]	-	-	-	7	4	3
sar reg8, cl	1101 0010 [11-111-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
sar reg16, cl	1101 0011 [11-111-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
sar reg32, cl	0110 0110 1101 0011 [11-111-r/m]	-	-	-	3	3	4
sar mem8, cl	1101 0010 [mod-111-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
sar mem16, cl	1101 0011 [mod-111-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
sar mem32, cl	0110 0110 1101 0011 [mod-111-r/m]	-	-	-	7	4	4
sar reg8, imm8	1100 0000 [11-111-r/m] [imm8]	-	-	5+imm8	3	2	1
sar reg16, imm8	1100 0001 [11-111-r/m] [imm8]	-	-	5+imm8	3	2	1
sar reg32, imm8	0110 0110 1100 0001 [11-111-r/m] [imm8]	-	-	-	3	2	1
sar mem8, imm8	1100 0000 [mod-111-r/m] [imm8]	-	-	8+imm8	7	4	3

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
sar mem16, imm8	1100 0001 [mod-111-r/m] [imm8]	-	-	8+imm8	7	4	3
sar mem32, imm8	0110 0110 1100 0001 [mod-111-r/m] [imm8]	-	-	-	7	4	3
sbb reg8, reg8	0001 10x0 [11-reg-r/m]	3	‘3	2	2	1	1
sbb reg16, reg16	0001 10x1 [11-reg-r/m]	3	3	2	2	1	1
sbb reg32, reg32	0110 0110 0001 10x1 [11-reg-r/m]	3	3	2	2	1	1
sbb reg8, mem8	0001 1010 [mod-reg-r/m]	9+EA	9+EA	7	7	2	2
sbb reg16, mem16	0001 1011 [mod-reg-r/m]	13+EA	9+EA	7	7	2	2
sbb reg32, mem32	0110 0110 0001 1011 [mod-reg-r/m]	-	-	-	7	2	2
sbb mem8, reg8	0001 1000 [mod-reg-r/m]	16+EA	16+EA	7	6	3	3
sbb mem16, reg16	0001 1001 [mod-reg-r/m]	24+EA	16+EA	7	6	3	3
sbb mem32, reg32	0110 0110 0001 1001 [mod-reg-r/m]	-	-	-	6	3	3
sbb reg8, imm8	1000 00x0 [11-011-r/m] [imm]	4	4	3	2	1	1
sbb reg16, imm16	1000 00s1 [11-011-r/m] [imm]	4	4	3	2	1	1
sbb reg32, imm32	0110 0110 1000 00s1 [11-011-r/m] [imm]	4	4	3	2	1	1
sbb mem8, imm8	1000 00x0 [mod-011-r/m] [imm]	17+EA	17+EA	7	7	3	3
sbb mem16, imm16	1000 00s1 [mod-011-r/m] [imm]	25+EA	17+EA	7	7	3	3
sbb mem32, imm32	0110 0110 1000 00s1 [mod-011-r/m] [imm]	-	-	-	7	3	3
sbb al, imm	0001 1100 [imm]	4	4	3	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
sbb ax, imm	0001 1101 [imm]	4	4	3	2	1	1
sbb eax, imm	0110 0110 0001 1101 [imm]	-	-	-	2	1	1
scasb	1010 0100	15	15	7	8	6	4
scasw	1010 0101	19	15	7	8	6	4
scasd	0110 0110 1010 0101	-	-	-	8	6	4
rep scasb	1111 0010 1010 0100	9 + 15 * cx	9 + 15*cx	5 + 8*cx	5 + 8*cx	7 + 5*cx 5 if cx=0	9 + 4*cx 7 if cx=0
rep scasw	1111 0010 1010 0101	9 + 19 * cx	9 + 15*cx	5 + 8*cx	5 + 8*cx	7 + 5*cx 5 if cx=0	9 + 4*cx 7 if cx=0
rep scasd	0110 0110 1111 0010 1010 0101	-	-	-	5 + 8*cx	7 + 5*cx 5 if cx=0	9 + 4*cx 7 if cx=0
seta reg8	0000 1111 1001 0111 [11-000-r/m][e]	-	-	-	4	4 if set 3 if clear	1
seta mem8	0000 1111 1001 0011 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setae reg8	0000 1111 1001 0011 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setae mem8	0000 1111 1001 0011 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setb reg8	0000 1111 1001 0010 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setb mem8	0000 1111 1001 0010 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setbe reg8	0000 1111 1001 0110 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setbe mem8	0000 1111 1001 0110 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setc reg8	0000 1111 1001 0010 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setc mem8	0000 1111 1001 0010 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
sete reg8	0000 1111 1001 0100 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
sete mem8	0000 1111 1001 0100 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setg reg8	0000 1111 1001 1111 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setg mem8	0000 1111 1001 1111 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setge reg8	0000 1111 1001 1101 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setge mem8	0000 1111 1001 1101 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setl reg8	0000 1111 1001 1100 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setl mem8	0000 1111 1001 1100 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setle reg8	0000 1111 1001 1110 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setle mem8	0000 1111 1001 1110 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setna reg8	0000 1111 1001 0110 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setna mem8	0000 1111 1001 0110 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnae reg8	0000 1111 1001 0010 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnae mem8	0000 1111 1001 0010 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnb reg8	0000 1111 1001 0011 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnb mem8	0000 1111 1001 0011 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnbe reg8	0000 1111 1001 0111 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnbe mem8	0000 1111 1001 0111 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
setnc reg8	0000 1111 1001 0011 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnc mem8	0000 1111 1001 0011 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setne reg8	0000 1111 1001 0101 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setne mem8	0000 1111 1001 0101 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setng reg8	0000 1111 1001 1110 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setng mem8	0000 1111 1001 1110 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnge reg8	0000 1111 1001 1100 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnge mem8	0000 1111 1001 1100 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnl reg8	0000 1111 1001 1101 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnl mem8	0000 1111 1001 1101 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnle reg8	0000 1111 1001 1111 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnle mem8	0000 1111 1001 1111 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setno reg8	0000 1111 1001 0001 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setno mem8	0000 1111 1001 0001 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnp reg8	0000 1111 1001 1011 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnp mem8	0000 1111 1001 1011 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setns reg8	0000 1111 1001 1001 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
setns mem8	0000 1111 1001 1001 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setnz reg8	0000 1111 1001 0101 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setnz mem8	0000 1111 1001 0101 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
seto reg8	0000 1111 1001 0000 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
seto mem8	0000 1111 1001 0000 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setp reg8	0000 1111 1001 1010 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setp mem8	0000 1111 1001 1010 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setpe reg8	0000 1111 1001 1010 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setpe mem8	0000 1111 1001 1010 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setpo reg8	0000 1111 1001 1011 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setpo mem8	0000 1111 1001 1011 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
sets reg8	0000 1111 1001 1000 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
sets mem8	0000 1111 1001 1000 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
setz reg8	0000 1111 1001 0100 [11-000-r/m]	-	-	-	4	4 if set 3 if clear	1
setz mem8	0000 1111 1001 0100 [mod-000-r/m]	-	-	-	5	3 if set 4 if clear	2
shl reg8, 1	1101 0000 [11-100-r/m]	2	2	2	3	3	1
shl reg16, 1	1101 0001 [11-100-r/m]	2	2	2	3	3	1
shl reg32, 1	0110 0110 1101 0001 [11-100-r/m]	-	-	-	3	3	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
shl mem8, 1	1101 0000 [mod-100-r/m]	15+EA	15+EA	7	7	4	3
shl mem16, 1	1101 0001 [mod-100-r/m]	23+EA	15+EA	7	7	4	3
shl mem32, 1	0110 0110 1101 0001 [mod-100-r/m]	-	-	-	7	4	3
shl reg8, cl	1101 0010 [11-100-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
shl reg16, cl	1101 0011 [11-100-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
shl reg32, cl	0110 0110 1101 0011 [11-100-r/m]	-	-	-	3	3	4
shl mem8, cl	1101 0010 [mod-100-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
shl mem16, cl	1101 0011 [mod-100-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
shl mem32, cl	0110 0110 1101 0011 [mod-100-r/m]	-	-	-	7	4	4
shl reg8, imm8	1100 0000 [11-100-r/m] [imm8]	-	-	5+imm8	3	2	1
shl reg16, imm8	1100 0001 [11-100-r/m] [imm8]	-	-	5+imm8	3	2	1
shl reg32, imm8	0110 0110 1100 0001 [11-100-r/m] [imm8]	-	-	-	3	2	1
shl mem8, imm8	1100 0000 [mod-100-r/m] [imm8]	-	-	8+imm8	7	4	3
shl mem16, imm8	1100 0001 [mod-100-r/m] [imm8]	-	-	8+imm8	7	4	3
shl mem32, imm8	0110 0110 1100 0001 [mod-100-r/m] [imm8]	-	-	-	7	4	3
shld reg16, reg16, imm8 r/m is 1st operand, reg is second operand.	0000 1111 1010 0100 [11-reg-r/m] [imm8]	-	-	-	3	2	4
shld reg32, reg32, imm8 r/m is 1st operand, reg is second operand.	0110 0110 0000 1111 1010 0100 [11-reg-r/m] [imm8]	-	-	-	3	2	4

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
shld mem16, reg16, imm8	0000 1111 1010 0100 [mod-reg-r/m] [imm8]	-	-	-	7	3	4
shld mem32, reg32, imm8	0110 0110 0000 1111 1010 0100 [mod-reg-r/m] [imm8]	-	-	-	7	3	4
shld reg16, reg16, cl r/m is 1st operand, reg is second operand.	0000 1111 1010 0101 [11-reg-r/m]	-	-	-	3	3	4
shld reg32, reg32, cl r/m is 1st operand, reg is second operand.	0110 0110 0000 1111 1010 0101 [11-reg-r/m]	-	-	-	3	3	4
shld mem16, reg16, cl	0000 1111 1010 0101 [mod-reg-r/m]	-	-	-	7	4	5
shld mem32, reg32, cl	0110 0110 0000 1111 1010 0101 [mod-reg-r/m]	-	-	-	7	4	5
shr reg8, 1	1101 0000 [11-101-r/m]	2	2	2	3	3	1
shr reg16, 1	1101 0001 [11-101-r/m]	2	2	2	3	3	1
shr reg32, 1	0110 0110 1101 0001 [11-101-r/m]	-	-	-	3	3	1
shr mem8, 1	1101 0000 [mod-101-r/m]	15+EA	15+EA	7	7	4	3
shr mem16, 1	1101 0001 [mod-101-r/m]	23+EA	15+EA	7	7	4	3
shr mem32, 1	0110 0110 1101 0001 [mod-101-r/m]	-	-	-	7	4	3
shr reg8, cl	1101 0010 [11-101-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
shr reg16, cl	1101 0011 [11-101-r/m]	8 + 4*cl	8 + 4*cl	5 + cl	3	3	4
shr reg32, cl	0110 0110 1101 0011 [11-101-r/m]	-	-	-	3	3	4
shr mem8, cl	1101 0010 [mod-101-r/m]	20+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
shr mem16, cl	1101 0011 [mod-101-r/m]	28+EA+4*cl	20+EA+4*cl	8 + cl	7	4	4
shr mem32, cl	0110 0110 1101 0011 [mod-101-r/m]	-	-	-	7	4	4

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
shr reg8, imm8	1100 0000 [11-101-r/m] [imm8]	-	-	5+imm8	3	2	1
shr reg16, imm8	1100 0001 [11-101-r/m] [imm8]	-	-	5+imm8	3	2	1
shr reg32, imm8	0110 0110 1100 0001 [11-101-r/m] [imm8]	-	-	-	3	2	1
shr mem8, imm8	1100 0000 [mod-101-r/m] [imm8]	-	-	8+imm8	7	4	3
shr mem16, imm8	1100 0001 [mod-101-r/m] [imm8]	-	-	8+imm8	7	4	3
shr mem32, imm8	0110 0110 1100 0001 [mod-101-r/m] [imm8]	-	-	-	7	4	3
shrd reg16, reg16, imm8 r/m is 1st operand, reg is second operand.	0000 1111 1010 1100 [11-reg-r/m] [imm8]	-	-	-	3	2	4
shrd reg32, reg32, imm8 r/m is 1st operand, reg is second operand.	0110 0110 0000 1111 1010 1100 [11-reg-r/m] [imm8]	-	-	-	3	2	4
shrd mem16, reg16, imm8	0000 1111 1010 1100 [mod-reg-r/m] [imm8]	-	-	-	7	3	4
shrd mem32, reg32, imm8	0110 0110 0000 1111 1010 1100 [mod-reg-r/m] [imm8]	-	-	-	7	3	4
shrd reg16, reg16, cl r/m is 1st operand, reg is second operand.	0000 1111 1010 1101 [11-reg-r/m]	-	-	-	3	3	4
shrd reg32, reg32, cl r/m is 1st operand, reg is second operand.	0110 0110 0000 1111 1010 1101 [11-reg-r/m]	-	-	-	3	3	4
shrd mem16, reg16, cl	0000 1111 1010 1101 [disp]	-	-	-	7	4	5
shld mem32, reg32, cl	0110 0110 0000 1111 1010 1101 [mod-reg-r/m]	-	-	-	7	4	5

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
stc	1111 1001	2	2	2	2	2	2
std	1111 1101	2	2	2	2	2	2
sti	1111 1011	2	2	2	3	5	7
stosb	1010 1010	11	11	3	4	5	3
stosw	1010 1011	15	11	3	4	5	3
stosd	0110 0110 1010 1011	-	-	-	4	5	3
rep stosb	1111 0010 1010 1010	9 + 10 * cx	9 + 10*cx	4 + 3*cx	5 + 5*cx	7 + 5*cx 5 if cx=0	9 + 3*cx 6 if cx=0
rep stosw	1111 0010 1010 1011	9 + 14 * cx	9 + 10*cx	4 + 3*cx	5 + 5*cx	7 + 5*cx 5 if cx=0	9 + 3*cx 6 if cx=0
rep stosd	0110 0110 1111 0010 1010 1011	-	-	-	5 + 5*cx	7 + 5*cx 5 if cx=0	9 + 3*cx 6 if cx=0
sub reg8, reg8	0010 10x0 [11-reg-r/m]	3	‘3	2	2	1	1
sub reg16, reg16	0010 10x1 [11-reg-r/m]	3	3	2	2	1	1
sub reg32, reg32	0110 0110 0010 10x1 [11-reg-r/m]	3	3	2	2	1	1
sub reg8, mem8	0010 1010 [mod-reg-r/m]	9+EA	9+EA	7	7	2	2
sub reg16, mem16	0010 1011 [mod-reg-r/m]	13+EA	9+EA	7	7	2	2
sub reg32, mem32	0110 0110 0010 1011 [mod-reg-r/m]	-	-	-	7	2	2
sub mem8, reg8	0010 1000 [mod-reg-r/m]	16+EA	16+EA	7	6	3	3
sub mem16, reg16	0010 1001 [mod-reg-r/m]	24+EA	16+EA	7	6	3	3
sub mem32, reg32	0110 0110 0010 1001 [mod-reg-r/m]	-	-	-	6	3	3
sub reg8, imm8	1000 00x0 [11-101-r/m] [imm]	4	4	3	2	1	1
sub reg16, imm16	1000 00s1 [11-101-r/m] [imm]	4	4	3	2	1	1
sub reg32, imm32	0110 0110 1000 00s1 [11-101-r/m] [imm]	4	4	3	2	1	1
sub mem8, imm8	1000 00x0 [mod-101-r/m] [imm]	17+EA	17+EA	7	7	3	3

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
sub mem16, imm16	1000 00s1 [mod-101-r/m] [imm]	25+EA	17+EA	7	7	3	3
sub mem32, imm32	0110 0110 1000 00s1 [mod-101-r/m] [imm]	-	-	-	7	3	3
sub al, imm	0010 1100 [imm]	4	4	3	2	1	1
sub ax, imm	0010 1101 [imm]	4	4	3	2	1	1
sub eax, imm	0110 0110 0010 1101 [imm]	-	-	-	2	1	1
test reg8, reg8	1000 0100 [11-reg-r/m]	3	‘3	2	2	1	1
test reg16, reg16	1000 0101 [11-reg-r/m]	3	3	2	2	1	1
test reg32, reg32	0110 0110 1000 0101 [11-reg-r/m]	3	3	2	2	1	1
test reg8, mem8	1000 0110 [mod-reg-r/m]	9+EA	9+EA	6	5	2	2
test reg16, mem16	1000 0111 [mod-reg-r/m]	13+EA	9+EA	6	5	2	2
test reg32, mem32	0110 0110 1000 0111 [mod-reg-r/m]	-	-	-	5	2	2
test reg8, imm8	1111 0110 [11-000-r/m] [imm]	4	4	3	2	1	1
test reg16, imm16	1111 0111 [11-000-r/m] [imm]	4	4	3	2	1	1
test reg32, imm32	0110 0110 1111 0111 [11-000-r/m] [imm]	4	4	3	2	1	1
test mem8, imm8	1111 0110 [mod-000-r/m] [imm]	9+EA	9+EA	6	5	2	2
test mem16, imm16	1111 0111 [mod-000-r/m] [imm]	13+EA	9+EA	6	5	2	2
test mem32, imm32	0110 0110 1111 0111 [mod-000-r/m] [imm]	-	-	-	5	2	2
test al, imm	1010 1000 [imm]	4	4	3	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
test ax, imm	1010 1001 [imm]	4	4	3	2	1	1
test eax, imm	0110 0110 1010 1001 [imm]	-	-	-	2	1	1
xadd reg8, reg8 r/m is first operand, reg is second operand.	0000 1111 1100 0000 [11-reg-r/m]	-	-	-	-	3	3
xadd reg16, reg16	0000 1111 1100 0001 [11-reg-r/m]	-	-	-	-	3	3
xadd reg32, reg32	0110 0110 0000 1111 1100 0001 [11-reg-r/m]	-	-	-	-	3	3
xadd mem8, reg8	0000 1111 1100 0000 [mod-reg-r/m]	-	-	-	-	4	4
xadd mem16, reg16	0000 1111 1100 0001 [mod-reg-r/m]	-	-	-	-	4	4
xadd mem32, reg32	0110 0110 0000 1111 1100 0001 [mod-reg-r/m]	-	-	-	-	4	4
xchg reg8, reg8	1000 0110 [11-reg-r/m]	4	4	3	3	3	3
xchg reg16, reg16	1000 0111 [11-reg-r/m]	4	4	3	3	3	3
xchg reg32, reg32	0110 0110 1000 0111 [11-reg-r/m]	-	-	-	3	3	3
xchg mem8, reg8[f]	1000 0110 [11-reg-r/m]	17 + EA	17 + EA	5	5	5	3
xchg mem16, reg16	1000 0111 [11-reg-r/m]	25 + EA	17 + EA	5	5	5	3
xchg mem32, reg32	0110 0110 1000 0111 [11-reg-r/m]	-	-	-	5	5	3
xchg ax, reg16	1001 0rrr	3	3	3	3	3 1 if reg=ax	2 1 if reg=ax
xchg ax, reg32	0110 0110 1001 0rrr	3	3	3	3	3	2
xlat	1101 0111	11	11	5	5	4	4
xor reg8, reg8	0011 00x0 [11-reg-r/m]	3	‘3	2	2	1	1
xor reg16, reg16	0011 00x1 [11-reg-r/m]	3	3	2	2	1	1

Table 97: 80x86 Instruction Set Reference[a]

Instruction	Encoding (bin)[b]	Execution Time in Cycles[c]					
		8088	8086	80286	80386	80486	Pentium
xor reg32, reg32	0110 0110 0011 00x1 [11-reg-r/m]	3	3	2	2	1	1
xor reg8, mem8	0011 0010 [mod-reg-r/m]	9+EA	9+EA	7	7	2	2
xor reg16, mem16	0011 0011 [mod-reg-r/m]	13+EA	9+EA	7	7	2	2
xor reg32, mem32	0110 0110 0011 0011 [mod-reg-r/m]	-	-	-	7	2	2
xor mem8, reg8	0011 0000 [mod-reg-r/m]	16+EA	16+EA	7	6	3	3
xor mem16, reg16	0011 0001 [mod-reg-r/m]	24+EA	16+EA	7	6	3	3
xor mem32, reg32	0110 0110 0011 0001 [mod-reg-r/m]	-	-	-	6	3	3
xor reg8, imm8	1000 00x0 [11-110-r/m] [imm]	4	4	3	2	1	1
xor reg16, imm16	1000 00s1 [11-110-r/m] [imm]	4	4	3	2	1	1
xor reg32, imm32	0110 0110 1000 00s1 [11-110-r/m] [imm]	4	4	3	2	1	1
xor mem8, imm8	1000 00x0 [mod-110-r/m] [imm]	17+EA	17+EA	7	7	3	3
xor mem16, imm16	1000 00s1 [mod-110-r/m] [imm]	25+EA	17+EA	7	7	3	3
xor mem32, imm32	0110 0110 1000 00s1 [mod-110-r/m] [imm]	-	-	-	7	3	3
xor al, imm	0011 0100 [imm]	4	4	3	2	1	1
xor ax, imm	0011 0101 [imm]	4	4	3	2	1	1
xor eax, imm	0110 0110 0011 0101 [imm]	-	-	-	2	1	1

a. Real mode, 16-bit segments.
b. Instructions with a 66h or 67h prefix are available only on 80386 and later processors.
c. Timings are all optimistic and do not include the cost of prefix bytes, hazards, fetching, misaligned operands, etc.
d. Cycle timings for HLT instruction are above and beyond the time spent waiting for an interrupt to occur.

e. On the 80386 and most versions of later processors, the processor ignores the *reg* field's value for the S*cc* instruction; the reg field, however, should contain zero.
f. Most assemblers accept "xchg reg,mem" and encode it as "xchg mem,reg" which does the same thing.

%OUT directive 424
(146
) 149
.LIST directive 425
.NOLIST directive 425
.RADIX 360
.XLIST directive 425
= Directive 362

16450/16550 serial communications chips 1223
80286 registers 148
80386 registers 149
8042 microcontroller chip 1154
80486 registers 149
80x86 registers 146
8250 registers 1224
8250 Serial Communications Chip 1223
8259A programmable interrupt controller 1005
8286 processor 99, 110
8486 processor 99, 116
8686 processor 99, 123
886 Processor 99, 110
90/10 rule 1311
90/10 rule, problems with using it 1312

A

AAA instruction 256, 258
AAD instruction 267
AAM instruction 264, 266
AAS instruction 259
Aborts 995
Absolute value (floating point) 796
Accepting states 887
Accessing a word in byte addressable memory 87
Accessing an element of a single dimension array 207
Accessing data with a 16-bit bus 89
Accessing double words in memory 91
Accessing elements of 3 & 4 dimensional arrays 213
Accessing elements of a two-dimensional array 212
Accessing elements of an array 209
Accessing elements of multidimensional arrays 217
Accessing fields of a structure 219
Accessing words at odd addresses 90
Accop routine (UCR Std Lib) 778
Accumulator register 99, 146
Acknowledge line 1200
Active modifiers 1155
Active TSRs 1029
ADC instruction 256
ADD instruction 195, 256
Add instruction sequence (x86) 108
Adders 61
Addition (extended precision) 470
Address binding 641, 642
Address bus 86
Address expressions 387
Address spaces 87
Addressable memory 86
Addressing modes 155, 387
Addressing modes (80x86) 162
Addressing modes (x86) 103
Adventure games 963
AH register 146
AL register 146
ALGOL 565
Algorithm 566
Algorithm implementation (optimizing) 1315
Alignment check flag 149
Allocating storage for arrays 216
Alt key status 293, 1168, 1358
AND 467
AND instruction 269
AND operation 20, 44
Anycset routine (UCR Std Lib) 915
APL 565
ARB routine (UCR Standard Library) 919
ARBNUM routine (UCR Std Lib) 920
Arccosecant 806
Arccosine 806
Arccotangent 806
Architecture 83
Arcsine 805
Arctangent 800
Arithmetic and logical unit (ALU) 100
Arithmetic expressions 460, 948
Arithmetic instructions 243, 255
Arithmetic logical systems 468
Arithmetic operations 459
Arithmetic operators in address expressions 388
Arithmetic shift right 27
Array access 207
Array implementation 207
Array initialization 208
Array variables 207
Arrays 206, 285
Arrays as structure fields 220
Arrays of arrays 213
Arrays of structures 220
Arrays of two or more dimensions 210
ASCII character set 15
Assembler directives 355
Assembler for the x86 processors 953
Assembling without linking 428
Assembly language header files 429
Assembly language statements 355
Assigning a constant to a variable 460
Assigning one variable to another 460
Assignments 460
Associativity 44, 463, 464

ASSUME directive 377
Asynchronous interrupts 997
AT (SEGMENT operand) 373
Atof (UCR Std Lib) 780
ATOH (UCR Std Lib) 341
ATOI (UCR Std Lib) 341
ATOU (UCR Std Lib) 341
Automata theory 883
Automaton 883
Autorepeat rate 293, 1168, 1358
Auxiliary flag 245
AX register 99, 146

B

Backtracking 890
Base (numbering system) specification 360
Base address (of an array) 207
Base address of a structure 219
Base pointer register 146
Base register 158
Base register (80386 & later) 164
Base(d) addressing mode 158
Based index plus displacement addressing (80386 & later) 164
Based indexed addressing (80386 & later) 164
Based indexed addressing mode 160
Based indexed plus displacment addressing mode 160
Basic System Components 83
Baud rate 1234
Baud rate (serial chip) 1225
BCD numbers 14
BH register 146
Biased (excess) exponents 775
Bidirectional parallel port 1199
Bidirectional parallel port data direction bit 1202
Bidrectional data transmission 1200
Big endian data format 254
binary 11
Binary coded decimal numbers 14
Binary constants 360
binary data types 14
Binary Formats 13
Binary Numbering System 12
Binary operator 43
Binding an address to a variable 642
BIOS keyboard support functions 1168
BIOS keyboard variables 1158
BIOS reentrancy problems 1033
Bit fields and packed data 28
Bit instructions 243, 269, 279
Bits 14
Bits per second (bps) 1225
BL register 146
Blurring a gray scale image 1317
Boolean Algebra 43
Boolean algebra 43
Boolean algebra theorems 44
Boolean expression canonical form 49
Boolean expressions 467
Boolean function equivalence to electronic circuits 59
Boolean function names 47
Boolean function numbers 47
Boolean function simplification 52
Boolean functions 45
Boolean functions of n variables 46
Boolean logical systems 468
Boolean map simplification 53
Boolean term 49
Boolean values 14
Boolean values represented as program states 469
BOUND instruction 292
Bounds exception 1001
BP register 146, 158
Branch out of range 297, 298
Break interrupt (serial chip) 1230
Break signal (serial chip) 1228
Breakpoint exception 1001
Brkcset routine (UCR Std Lib) 915
BSF instruction 279
BSR instruction 279
BSWAP instruction 252, 254
BT instruction 279
BTC instruction 279
BTR instruction 279
Bugs in macros 420
Bus contention 118
Bus interface unit (BIU) 100
Busy line (parallel port) 1200
BX register 146, 158
Byte 14
Byte addressable memory array 88
Byte directive 384
Byte enable lines 87, 91
BYTE pseudo-opcode 199
BYTE PTR operator 390
Byte strings 819
Byte variables 198
BYTE variables, initialized 200
Bytes 13

C

C strings 831
C/C++ 565
Cache and its effects on performance 119
Cache hit 97
Cache hit ratio 98
Cache memory 96
Cache miss 97
Cache, two level 98
Calculator application 948
CALL instruction 289, 566
Callee register preservation 573
Caller register preservation 573

Canonical forms 49
Capslock 1155
Capslock key status 293, 1168, 1358
Carry flag 244, 302
Case labels (non-contiguous) 527
Case Statement 525
Case statement 522
CBW instruction 252
CDQ instruction 252
Central Processing Unit 83
CH register 146
Chaining interrupt service routines 1010
Change sign (floating point) 797
Changing the type of a symbol 390
Character constants 361
Character set 854
Character string functions 835
Choosing better algorithms 1315
Church's hypothesis 883
CISC 166
CL register 146
CLASS type (SEGMENT operand) 374
Classifying characters for a DFA/state machine 897
CLC instruction 302
CLD instruction 302
Clear to send (CTS) signal on the serial port 1230
Clearing the FPU exception bits 801
CLI instruction 302
Clock 92
Clock frequency 93
Clock period 93
Clocked logic 62
Closure 43
Closure of an operator 43
CMC instruction 302
CMP instruction 263
CMPS 819, 826
CMPS instruction 284
CMPXCHG instruction 263
Code stream parameters 574
Codeview support for floating point variables 202
Codeview support for SWORD/WORD 201
Coercion 390, 472
Column major ordering 211, 215
COM port addresses 1223
COM1
, COM2
, COM3
, and COM4
ports 1223
ComBaud routine (Standard Library) 1231
Combinatorial circuits 60
Combine type (SEGMENT operand) 373
ComDisIntr routine (Standard Library) 1232
ComGetIER routine (Standard Library) 1232
ComGetIIR routine (Standard Library) 1232
ComGetLCR routine (Standard Library) 1232
ComGetLSR routine (Standard Library) 1232
ComGetMCR routine (Standard Library) 1232
ComGetMSR routine (Standard Library) 1232
ComIn routine (Standard Library) 1232
ComInitIntr routine (Standard Library) 1232
Comment field 356
COMMON (SEGMENT operand) 373
Commutative operators 466
Commutativity 43
ComOut routine (Standard Library) 1232
Compare strings 819
Comparing floating point numbers 773
Comparing floating point values 780
Comparing pointers 154
Comparing strings 848
Comparison of strings 834
ComParity routine (Standard Library) 1231
Compile-only assembly 428
Complex expressions 462
Complex string functions 830
Composite data types 206
Computer Architecture 83
Computing 10**x 807
Computing 2**x 795, 799, 807
Computing LN(x) 808
Computing LOG(x) (base 10) 808
Computing Y**X 808
ComRead routine (Standard Library) 1231
ComSetIER routine (Standard Library) 1232
ComSetLCR routine (Standard Library) 1232
ComSetMCR routine (Standard Library) 1232
ComSize routine (Standard Library) 1231
ComStop routine (Standard Library) 1231
ComTstIn routine (Standard Library) 1232
ComTstOut routine (Standard Library) 1232
ComWrite routine (Standard Library) 1232
Concatenation 847
Concatenation (string function) 844
Condition codes 244
Conditional assembly 397
Conditional jump aliases 298
Conditional jump instructions 296
Conditional jump out of range 297
Conditional jumps (x86) 106
Constants 359
Constructing a truth map 53
Constructing logic functions using only NAND operations 59
Constructing patterns for the match routine (UCR Std Lib) 933
Constructing truth tables from the canonical form 49
Contention (for the bus) 118
Context free grammar 900
Context free languages 900
context free languages 884
Control bus 86
Control characters 29
Control key status 293, 1168, 1358
Control register (parallel port) 1201
Control Structures 521
Control unit (CU) 100

Conversion instructions 252
Conversions 243
Converting a DFA to assembly language 895
Converting a string to upper or lower case 852
Converting BCD to floating point 792
Converting between canonical forms 52
Converting binary to hex 18
Converting CFGs to assembly language 905
Converting CFGs to Std Lib patterns 933
Converting dates in English to integers 941
Converting hex to binary 18
Converting integers to floating point 791
Converting numbers in English to integers 935
Converting REs to CFGs 905
Coprocessor unavailable exception 1004
Copying strings 849
Cosecant 805
Cosine 799
Cotangent 805
Count (string elements) 820
Counters 64
CPU 83
CPU Registers 99
Critical region/section 1013
CS register 155
CWD instruction 252
CWDE instruction 252
Cycle counting 1315

D

D (data) flip-flop 63
DAA instruction 256, 258
DAS instruction 259
Data available on the serial chip 1229
Data bus 84
Data carrier detect (DCD) signal on the serial chip 1231
Data direction bit (bidirectional parallel port) 1202
Data movement instructions 243
Data register (parallel port) 1201
Data register (serial chip) 1224
Data set ready (DSR) signal on the serial chip 1230
Data terminal ready (DTR) signal on the serial port 1228
Dates (DOS) 718
DB directive 384
DB pseudo-opcode 199
DD directive 384
DD pseudo-opcode 201
Deactivating ctrl-alt-del 1184
Debug resume flag 149
Debugging code with IFDEF 399
Debugging registers 149, 1001
DEC instruction 259
decimal 11
Decimal constants 360
Decision 521
Declaring arrays 207
Declaring byte variables 198
Declaring variables 196
Declaring your own types 203
Decoding an instruction 107
Default numeric base 360
Default segment for memory addressing mode (80x86) 168
Default segment in addressing mode (80386) 165
Defining a macro 400
Delay Loops 544
Delete (string function) 843
Deleting characters from a string 850
Deleting leading spaces 846
Deleting trailing spaces from a string 855
Denormalized exception (FPU) 784
Denormalized values 777
Derivation 902
Destination index 820
Destination index register 158
Deterministic finite state automata 884, 893
DH register 146
DI register 158
Direct addressing mode 156
Direct memory access 124
Direction flag 244, 285, 820, 821
Disabling interrupts 1006
Disassembly 130
Disk drive interrupt 1009
Disk transfer area 1040
Displacement only addressing mode 156
Displacement only MOD-REG-R/M byte encoding 168
Display (lexical nesting data structure) 639
Distributive law 44
DIV instruction 267
Divide error exception 1000
Divide errors 268
Division instructions 267
DL register 146
DMA 124
Domain conditioning 496
Domain of a function 494
DOS Idle interrupt 1033
DOS reentrancy problems 1032
DOS' free memory pointer 1025
Dot operator 219
Double precision floating point format 776
Double precision shift instructions 270, 274
Double word storage in byte addressable memory 87
Double word strings 819
Double words 16
Down key code 1153
DQ directive 384
DS register 155
DT directive 384
Duality 45
DUP operator 207
Duplicating strings 849
DW directive 384
DW pseudo-opcode 200

DWORD directive 384
DWORD pseudo-opcode 201
DWORD PTR operator 390
Dynamic link 643, 666
Dynamically allocated strings 831
Dynamically assigning TSR identifiers 1035

E

Early optimization 1311
EAX register 149
EBP register 149
EBX register 149
ECHO directive 424
ECX register 149
EDI register 149
EDX register 149
Effective address 162, 249
Efficency of macros 419
EFLAGS register 149
Eight-bit register 146
EIP register 149
Electronic circuit equivalence to boolean functions 59
Eliminating left recursion 903
ELSE 522
ELSE directive 398
Enabling interrupts 1006
Enabling interrupts on the 8250 serial chip 1229
Encoding for the displacement only addressing mode 168
End of file 334
End of interrupt signal (8259) 1006
ENDIF directive 398
ENDP directive 566
Enter instruction 249
EOS routine (UCR Std Lib) 919
EQU directive 362
Equates 362
ES register 155
ESI register 149
ESP register 149
Etoa (UCR Std Lib) 780
Evaluating arithmetic expressions 948
Even parity 1228
Exception flags (FPU) 785
Exception masks (FPU) 784
Exceptions 995, 1000
Exclusive-or 20
Exclusive-OR operation 47
Exclusive-or operation 21
Execution units 123
EXITM directive 406
Exp(x) (e**x) 807
Exponent 772
Expressions 460
Expressions and temporary values 466
Extended addressing 151
Extended error global data (DOS) 1041
Extended keyboard codes 1155
Extended keyboard status 294, 1169, 1359
Extended precision addition 470
Extended precision floating point format 776
EXTERN types 427
EXTERN/EXTRN directives 427
EXTERNDEF directive 428
Extracting substrings from matched patterns 925

F

F2XM1 instruction 799
FABS instruction 796
Fadd (UCR Std Lib) 780
FADD/FADDP instructions 792
Failure state 894
Falling edge of a clock 93
False (representation) 467
Far calls 391
Far jump instructions 287
Far pointers 205
Far procedures 365, 568
FAR PTR operator 390
Far return 569
Faults 995
FBLD/FBSTP instructions 792
FCHS instruction 797
FCLEX/FNCLEX instructions 801
Fcmp (UCR Std Lib) 780
FCOM/FCOMP/FCOMPP instructions 797
FCOS instruction 799
FDECSTP instruction 803
Fdiv (UCR Std Lib) 780
FDIV/FDIVP/FDIVR/FDIVRP instructions 794
Fetching an opcode 107
FFREE instruction 803
FIADD instruction 803
FICOM instruction 803
FICOMP instruction 803
FIDIV instruction 803
FIDIVR instruction 803
FILD instruction 791
FIMUL instruction 803
Final states 887
FINCSTP instruction 803
FINIT/FNINIT instructions 800
FIST/FISTP instructions 791
FISUB instruction 803
FISUBR instruction 803
Flags 244
Flags (and CMP) 261
Flags register 148
Flat addressing 151
FLD instruction 789
FLD1 instruction (load 1.0) 798
FLDCW instruction 801
FLDENV instruction 801

FLDL2E instruction (load lg(e)) 798
FLDL2T instruction (load lg(10)) 798
FLDLG2 instruction (load log(2)) 798
FLDLN2 instruction (load ln(2)) 798
FLDPI instruction (load pi) 798
FLDZ instruction (load 0.0) 798
Flip-flops 62
Floating point - integer conversions 779
Floating point arithmetic 771
Floating point comparisons 252, 773, 797
Floating point constants 202
Floating point control register 782
Floating point coprocessors 781
Floating point routines (UCR Std Lib) 777
Floating point values 17
Floating point variables 202
Floppy disk interrupt 1009
Flushing the pipeline 119
Fmul (UCR Std Lib) 780
FMUL/FMULP instructions 794
FNOP instruction 803
FOR directive 420
For loops 533
FORC directive 420
Forcing bits to one 22
Forcing bits to zero 22
Formal language theory 883
FORTH 565
FORTRAN 565
FPATAN instruction 800
FPREM/FPREM1 instructions 795
FPTAN instruction 799
FPU busy bit 788
FPU condition code bits 785
FPU control word 801
FPU environment record 801
FPU exception bits 801
FPU exception flags 785
FPU exception masks 784
FPU interrupt 1009
FPU interrupt enable mask 784
FPU precision control 784
FPU stack fault flag 785
FPU Stack pointer 803
FPU Status register 803
FPU status register 785
FPU top of stack pointer 788
FPUs 781
Framing errors (serial chip) 1230
Free (UCR Std Lib) 334
Free memory pointer 1025
Frequency of interrupts 1015
FRNDINT instruction 796
FRSTOR instruction 802
FS register 155
FSAVE/FNSAVE instructions 802
FSCALE instruction 795
FSIN instruction 799
FSINCOS instruction 799
FSQRT instruction 795
FST/FSTP instructions 790
FSTCW instruction 801
FSTENV/FNSTENV instructions 801
FSTSW/FNSTSW instructions 803
Fsub (UCR Std Lib) 780
FSUB/FSUBP/FSUBR/FSUBRP instructions 793
Ftoa (UCR Std Lib) 780
Ftoi (UCR Std Lib) 779
Ftol (UCR Std Lib) 779
Ftou (UCR Std Lib) 779
Ftoul (UCR Std Lib) 779
FTST instruction 798
FUCOM/FUCOMP/FUCOMPP instructions 798
Full adders 61
Function instance 642
Function numbers 47
Function results 600
Functional units 110
Functions 565, 572
FWAIT instruction 801
FWORD pseudo-opcode 202
FXCH instruction 790
FXTRACT instruction 796
FYL2X instruction 800
FYL2XP1 instruction 800

G

Games 963
Garbage collection 831
General purpose registers 146
Generating tables 497
Generic MOV instruction 166
Get date (DOS) 718
Get interrupt vector call (DOS) 998
Get time (DOS) 718
GETC (UCR Std Lib) 334
GETS (UCR Std Lib) 334
GETSM (UCR Std Lib) 334
Global memory locations as parameters 574
GotoPos routine (UCR Std Lib) 921
GS register 155
Guard digits/bits 772

H

H.O. 13
Half adder 61
Handling reentrancy in DOS 1032
Handshaking 1200
Handshaking (serial chip) 1228
Hardware interrupts 995, 1004
Hardware stack operation 251
Harvard architecture 120
Hazards 122

Header files 429
Heap 334
Hertz (Hz) 93
Hexadecimal 14
hexadecimal 11
Hexadecimal Calculators 19
Hexadecimal calculators 19
Hexadecimal constants 360
Hexadecimal numbering system 17
HIGH operator 392
High order bit 13, 14
High order byte 16
High order nibble 15
High order word 16
HIGHWORD operator 392
HLT instruction 302
Hot keys 1184
Hot spots in code 1313
HTOA (UCR Std Lib) 341

I

I/O 124
I/O address bus 87
I/O instructions 243, 284
I/O mapped input/output 124
I/O port 124
I/O ports 284
I/O subsystem 92
ICON 565
Identity element for boolean operations 44
Identity elements 44
IDIV instruction 267
Idle interrupt 1033
IEEE floating point standard (754 & 854) 774
IF directive 398
IF..THEN..ELSE 521, 522
IFB directive 399
IFDEF directive 399
IFDIF directive 400
IFDIFI directive 400
IFE directive 399
IFIDN directive 400
IFIDNI directive 400
IFNB directive 399
IFNDEF directive 399
Implementing an algorithm better 1315
IMUL instruction 264
IMUL/MUL differences 266
IN instruction 284
INC instruction 256, 258
INCLUDE directive 426
Index (string function) 838
Index register 158
Index register (80386 & later) 164
Indexed addressing (80386 & later) 164
Indexed addressing (scaled) 165
Indexed addressing mode 158, 159
Indexed addressing mode (x86) 104
Indirect addressing mode 104
Indirect jump 531
Indirect jump instructions 287
Indirect jumps 522
InDOS flag 1032
Induction variables 540
Infinite precision arithmetic 771
Inhibition operation 47
Initializing a string 819
Initializing array variables 208
Initializing BYTE variables 200
Initializing fields of a structure 220
Initializing interrupt vector table entries 997
Initializing strings and arrays 829
Input conditioning 496
INS instruction 284
Insert (string function) 841
Insert key status 293, 1168, 1358
Inserting characters into a string 851
Inserting characters into the typeahead buffer 293, 1168, 1358
Installing a TSR 1035
Instance 642
Instruction encodings 245
Instruction pointer (IP) 148
Instruction pointer register 102
Instruction pointer register (IP) 99
Instruction prefixes 830
Instruction set 243
INT 0Bh 1008
INT 0Ch 1008
INT 0Dh 1008
INT 0Eh 1009
INT 0Fh 1008
INT 16h keyboard service routine 1169
INT 1Ch 1007
INT 75h 1009
INT 76h 1009
INT 8 1007
INT 9 1008
Int 9 (patching the keyboard interrupt) 1184
Int 9 interrupt service routine 1174
INT instruction 292
INT operation 295
Integer - floating point conversion 779
Integer constants 360
Integer division by two 27
Interrupt 995
Interrupt chaining 1010
Interrupt driven serial I/O 1239
Interrupt enable mask (FPU) 784
Interrupt enable on the 8250 serial chip 1229
Interrupt enable register (serial chip) 1224
Interrupt flag 244, 302
Interrupt frequency 1015
Interrupt identification register (serial chip) 1224
Interrupt in-service register (8259) 1007

Interrupt latency 1016
Interrupt latency consistency 1020
Interrupt mask register (8259) 1006
Interrupt priorities 1020
Interrupt request register (8259) 1007
Interrupt service routine 127, 995
Interrupt service routine (x86) 107
Interrupt service time 1015
Interrupt sources on the serial chip 1226
Interrupt vector 127
Interrupt vector table 996
Interrupts 126
Interrupts and reentrancy 1012
Intersegment jump instruction 286
INTO instruction 292
Intrasegment jump instructions 286
Invalid opcode exception 1004
Invalid operation exception (FPU) 784
Invariant computations 538
Inverse element 44
Inverse element for boolean operations 44
Inverting bits 22
Invoking a macro 401
IRET instruction 292
IRP directive 420
IRPC directive 420
ISR 127
ITOA (UCR Std Lib) 341
Itof (UCR Std Lib) 779

J

JA instruction 297
JAE instruction 297
JB instruction 297
JBE instruction 297
JC instruction 296
Jcc instructions 296
Jcc out of range 297
JCXZ instruction 299
JE instruction 297
JECXZ instruction 299
JG instruction 297
JGE instruction 297
JL instruction 297
JLE instruction 297
JMP instruction 286
JNA instruction 297
JNAE instruction 297
JNB instruction 297
JNBE instruction 297
JNC instruction 296
JNE instruction 297
JNG instruction 297
JNGE instruction 297
JNL instruction 297
JNLE instruction 297
JNO instruction 296
JNP instruction 296
JNS instruction 296
JNZ instruction 296
JO instruction 296
JP instruction 296
JPE instruction 296
JPO instruction 296
JS instruction 296
JZ instruction 296

K

Keyboard 1153
Keyboard controller command byte 1162
Keyboard interrupt service routine 1174
Keyboard interrupts 1008
Keyboard LEDs 1163
Keyboard microcontroller command set 1160
Keyboard microcontroller commands 1162
Keyboard microcontroller status 1160
Keyboard modifiers 1154
Keyboard scan code 1153
Keyboard scan codes 1156, 1351
Keyboard to system commands 1167
Keybounce 1153
Kleene Plus 886
Kleene Star 885
Kost significant bit 14

L

L.O. 13
Label field 355
Label format 358
Label types 385
Label values 386
Labels 358
LAHF instruction 252
Laplink 1209
Laplink parallel cable connections 1209
Large programs 425
Late optimization 1311
Latency (interrupts) 1016
Latency consistency 1020
Lazy evaluation 574
Ldfpa routine (UCR Std Lib) 778
Ldfpo routine (UCR Std Lib) 779
LDS instruction 248
LEA instruction 162, 195, 248
Leading spaces in a string 846
Least significant bit 14
Leave instruction 249
Lefpa routine (UCR Std Lib) 778
Lefpal routine (UCR Std Lib) 779
Lefpo routine (UCR Std Lib) 779
Lefpol routine (UCR Std Lib) 779

Left associative operators 464
Left factoring 903
Left recursive grammars 903
Left shift 26
Length of a string 852
LENGTH operator 392
Length prefixed strings 831
LENGTHOF operator 392
LES instruction 195, 248
Lexical Nesting 639
Lexicographical ordering 826
LFS instruction 248
LGS instruction 248
Lifetime of a variable 642
Line continuation symbol 395
Line control register (serial chip) 1224
Line status register (serial chip) 1224
Linear addressing 151
LISP 565
LIST (.LIST) directive 425
Listing directives 424
Literal constants 359
Literals (boolean) 49
Little endian data format 254
LN(x) 808
Load effective address instruction 248
Load instruction operation (x86) 107
Loading and storing floating point values 778
LOCAL directive (for macros) 406
Local variables 604
Locality of reference 96
Location counter 357, 367
LOCK prefix instruction 303
LODS 819, 829, 830
LODS instruction 284
LOG(x) (base 10) 808
Logarithms (base 2) 800
Logical addresses 152
Logical AND 44
Logical AND operation 20
Logical complement 44
Logical exclusive-OR 47
Logical exclusive-or operation 20, 21
Logical expressions 467
Logical inhibition 47
Logical instructions 243, 269
Logical NAND 47
Logical NOR 47
Logical NOT 47
Logical NOT operation 20, 22
Logical operations 459
Logical Operations on Binary Numbers 22
Logical Operations on Bits 20
Logical operators in address expressions 388
Logical OR 44
Logical OR operation 20, 21
Logical parallel port addresses 1202
Logical shift right 27
Logical to physical address translation (protected mode) 153
Logical to physical address translation (real mode) 152
Logical XOR operation 20
Loop 521
Loop control variables 532
LOOP instruction 534
Loop instruction 300
Loop invariant computations 538
Loop register usage 534
Loop termination 535
Loop termination test 532
Loop unraveling 539
Loop..Endloop 533
Loopback mode (serial chip) 1228
LOOPE/LOOPZ instruction 300
LOOPNE/LOOPNZ instruction 300
Loops 531
LOW operator 392
Low order bit 13, 14
Low order byte 16
Low order nibble 15
Low order word 16
Lower case conversion 852
LOWWORD operator 392
LPT1
, LPT2
, LPT3
ports 1199
Lsfpa routine (UCR Std Lib) 778
Lsfpo routine (UCR Std Lib) 779
LSS instruction 248
Ltof (UCR Std Lib) 779

M

Machine state, saving the 572
Macro operators 407
Macro parameter expansion 407
Macros 400, 404
Macros vs. procedures 404
Madventure 963
Make files 429
MALLOC (UCR Std Lib) 334
Managing large programs 425
Manifest constants 360, 362
Mantissa 772
Map method for boolean function simplification 53
Masking 23
Masking out 14
Masks 490
MASM reserved words 358
Matchchar routine (UCR Std Lib) 917
Matchchars routine (UCR Std Lib) 918
Matchistr routine (UCR Std Lib) 916
Matchstr routine (UCR Std Lib) 916
Matchtochar routine (UCR Std Lib) 918
Matchtopat routine (UCR Std Lib) 918

Matchtostr routine (UCR Std Lib) 917
Maximum addressable memory 86
Megahertz (Mhz) 93
MEMINIT (UCR Std Lib) 334
MEMORY (SEGMENT operand) 373
Memory access 93
Memory access time 93
Memory addressing modes (80386 & later) 163
Memory addressing, default segment 165
Memory banks 89
Memory cells 62
Memory management 151
Memory organization 150
Memory subsystem 87
Memory to memory moves 169
Memory usage under DOS 1025
Memory-mapped I/O 124
Merging source files during assembly 426
Metaware Professional Pascal 665
Microprocessor clock 92
Miscellaneous instructions 243
Mnemonic field 356
MOD field encodings in MOD-REG-R/M byte 167
Modem control register (serial chip) 1224
Modem status register (serial chip) 1224
Modifier key status 293, 1168, 1358
Modifier keys 1154
Modifying the FPU stack pointer 803
MOD-REG-R/M byte 166
MOD-REG-R/M encoding for R/M field 168
MOD-REG-R/M Reg field encodings 167
Modular design 565
Modules 565
Modulo (floating point remainder) 795
MOV instruction 156, 166, 246
MOV instruction encoding 166
Move strings 819
Moving data from one segment register to another 156
MOVS 819, 822
MOVS instruction 284
MOVSX instruction 252
MOVZX instruction 252
MUL instruction 195, 264
MUL/IMUL differences 266
Multidimensional arrays 210
Multiplex interrupt 1034
Multiplication instructions 264
Multiprecision addition 470
Multi-precision integers 859
Multitasking 1025

N

Names of boolean functions 47
NAND gates 59
NAND operation 47
Near jump instructions 287
Near pointers 204
Near procedures 365, 568
NEAR PTR operator 390
Near return 569
Near symbols 385
Nectored interrupts 996
NEG instruction 263
Negation 462
Negation (floating point) 797
Nested procedures 569
Nested statements and loops 542
Nested task flag 148
Newline 336
Nibble 14
Nibbles 13
Nmake.exe program 429
NOLIST (.NOLIST) directive 425
Nondeterministic Finite State Automata 887
Nondeterministic finite state automata 884
Nonmaskable Interrupts 1009
Nonvectored interrupts 996
NOP instruction 302
NOR operation 47
Normalized addresses 154
Normalized values 777
NOT 467
NOT instruction 269
NOT operation 20, 22, 44, 47
Notanycset routine (UCR Std Lib) 916
NOTHING (ASSUME operand) 378
Number of boolean functions 46
Numlock 1155
Numlock key status 293, 1168, 1358

O

Odd parity 1227
OFFSET operator 392
Offset portion of an address 151
Offsets, 16-bits 152
Offsets, 32-bits 152
OPATTR operator 392
Opcodes 102
Operand field 356
Operation codes 102
Operator precedence 396, 463
Opposite jumps 298
Optimal algorithms 1315
Optimization 1311
Optimization – three forms 1315
Optimization via cycle counting 1315
Optimization vs. fast hardware 1315
OR 20, 467
OR instruction 269
OR Operation 21
OR operation 44
OTHERWISE (in CASE) 526

OUT (%OUT) directive 424
OUT instruction 284
OUTS instruction 284
Overflow exception 1001
Overflow exception (FPU) 784
Overflow flag 244
Overlapping blocks (string operations) 823

P

Packed data 28
PAGE directive 424
Paragraph 369
Paragraph addresses 16
Parallel (printer) ports 1199
Parallel data transmission 1199
Parallel port acknowledge line 1200
Parallel port base address 1202
Parallel port data communications 1209
Parallel port data direction bit 1202
Parallel port data, status, and control registers 1201
Parallel port handshaking 1200
Parallel port interrupt 1008
Parallel port IRQ enable 1202
Parallel port signals 1201
Parallel port strobe line 1200
Parameters 291, 574
Parameters, variable length 592
Parity errors 1231, 1236
Parity errors (serial chip) 1230
Parity errors and the serial port 1227
Partial remainder 795
Pascal strings 831
Pass by lazy evaluation 574, 654
Pass by name 654
Pass by name parameters 574
Pass by reference 653
Pass by reference parameters 574
Pass by result 653
Pass by value 652
Pass by value parameters 574
Pass by value/returned 575
Pass by value/returned parameters 574
Pass by value-result 653
Passing control from one ISR to another 1010
Passing parameters by lazy-evaluation in a block structured language 654
Passing parameters by name 576
Passing parameters by name in a block structured language 654
Passing parameters by reference in a block structured language 653
Passing parameters by result 576
Passing parameters by Result in a block structured language 653
Passing parameters by value in a block structured language 652
Passing parameters by value-result in a block structured language 653
Passing parameters from one procedure as parameters to another 655
Passing parameters in a parameter block 574, 598
Passing parameters in global memory locations 574
Passing parameters in global variables 580
Passing parameters in registers 574, 578
Passing parameters in the code stream 574, 590
Passing parameters on the stack 574, 581
Passing variables from different lex levels as parameters 652
Passive TSRs 1029
Patch panel programming 101
Patching an application 1055
Patching the keyboard interrupt (int 9) 1184
Patgrab routine (UCR Std Lib) 926
Pattern data structure (UCR Std Lib) 913
Pattern matching 883
Pattern matching functions 922
Performance improvements for loops 535
Physical addresses 152
PIC 1005
Pipeline flush 119
Pipeline stalls 118
Pipelining 116
Pixel 1318
PL/I 565
Pointers 203
Pointers to structures 221
Polled I/O 126
Polling 1014
Polling the serial port 1236
POP instruction 249
POPA/POPAD instruction 249
POPF instruction 249
Pop-up programs 1029
Port 124
Port addresses 284
Pos routine (UCR Std Lib) 921
Precedence 396, 463
Precision exception (FPU) 784
Prefetch queue 112
Prefetch queue and effects on performance 119
Prefixes 830
Preserving registers 572
Principle of duality 45
PRINT (UCR Std Lib) 336
Printer device BIOS variables 1203
Printer time-out variables 1203
PRINTF (UCR Std Lib) 336
Printff (UCR Std Lib) 780
Printing a character 1203
Prioritized interrupts 1020
Problems with the 90/10 rule 1312
PROC directive 566
Procedural languages 565
Procedural macros 400
Procedure instance 642
Procedure invocation 566
Procedure standard entry code 582
Procedure standard exit code 582
Procedures 365, 565

Procedures vs. macros 404
Processor size 85
Processor status register 244
Product of maxterms representation 49
Professional Pascal 665
Profiler program 1313
Program analysis for optimization 1314
Program flow instructions 243, 286
Program memory usage under DOS 1025
Program unit 644
Programmable interrupt controller 1005
Programming in the large 426
PROLOG 565
Protected mode 152, 153
Protected mode instructions 303
PrtSc key and INT 5 1004
Pseudo opcodes 355
PSP 1040
PTR operator 390, 392
PUBLIC (SEGMENT operand) 373
PUBLIC directive 427
push down automata 884
PUSH instruction 249
PUSHA/PUSHAD instruction 249
Pushdown automata 902
PUSHF instruction 249
PUTC (UCR Std Lib) 336
PUTCR (UCR Std Lib) 336
PUTH (UCR Std Lib) 336
PUTI (UCR Std Lib) 336
Putisize routine (std lib) 336
PUTS (UCR Std Lib) 336
Putusize routine (std lib) 336

Q

Quicksort 607
QWORD directive 384
QWORD pseudo-opcode 202

R

radix 17
RADIX specification 360
Range of a function 494
RCL instruction 276, 277
RCR instruction 276, 277
Read control line 87
Reading a character from the keyboard 293, 1168, 1358
Reading characters from the keyboard (DOS) 1167
Reading data from the serial port 1231
Reading from memory 87
Real addresses 150
Real mode 150, 153
REAL10 pseudo-opcode 202
REAL4 pseudo-opcode 202
REAL8 pseudo-opcode 202
Recognizers 884
Records 218
Recursion 606
Reducing the size of a DFA/state machine table 897
Redundant instructions on 80x86 168
Reentrancy 1032
Reentrancy problems with the BIOS 1033
Reentrant programs 1012
REG field encoding of MOD-REG-R/M byte 168
REG field encodings in MOD-REG-R/M byte 167
Register addressing modes 156
Register addressing modes (80386 & later) 163
Register indirect addressing (80386 & later) 163
Register indirect addressing mode 158
Register preservation 572
Register usage in loops 534
Registers 146
Registers (electronic implementation) 63
Registers as procedure parameters 574, 578
Regular Expressions 885
regular languages 884
Relational operators in address expressions 388
Relocatable expressions 389
Remainder (floating point) 795
Removing a TSR 1037
Removing trailing spaces from a string 855
REP/REPE/REPZ/REPNE/REPNZ instructions 284
Repeat (string function) 840
REPEAT directive 420
Repeat Until loop 532
Repeating a character throughout a string 853
REPT directive 420
Request to send (RTS) on the serial port 1228
Reserved words 358
Reset (ctrl-alt-del) deactivation 1184
Resetting interrupt conditions on the serial chip 1226
Resident portion of a TSR 1026
Resident programs 999
Resume flag 149
Resume frame (for iterators) 666
RET instruction 289, 566
RETF instruction 569
RETN instruction 569
Reversing the characters in a string 853
RGotoPos routine (UCR Std Lib) 922
Right associative operators 464
Right shift 26
Ring indicator (RI) signal on the serial chip 1230
Rising edge of a clock 93
ROL instruction 276, 278
ROR instruction 276, 278
Rotate instructions 243, 269, 276
Rotate left 27
Rotate right 27
Rounding a floating point value to an integer 796
Rounding control (FPU) 783
Row major ordering 211
RPos routine (UCR Std Lib) 921

S

SAHF instruction 252
SAL instruction 270, 271
SAR instruction 270, 272
Saving FPU state 802
Saving the machine state 572
SBB instruction 259
Sbyte directive 384
SBYTE pseudo-opcode 199
Scalar variables 197
Scaled indexed addressing mode 165
Scan code 1153
SCAS 819, 828
SCAS instruction 284
SCC (serial communications chip) 1223
Schematic symbols 59
Scope 363, 639
Scroll lock 1155
Scroll lock key status 293, 1168, 1358
SDWORD directive 384
SDWORD pseudo-opcode 201
Search for a single character within a string 848
Searching for data within a string 819
Searching for one string within another 855
Secant 805
SEG operator 392
Segment loading order 368, 375
Segment names 367
Segment override prefix 157
Segment portion of an address 151
Segment prefixes 377
Segment registers 155
SEGMENT statement operands 369
Segmentation as a two-dimensional access 152
Segmented address 16
Segmented addresses 152
Segments 366
Segments on the 80x86 151
Self-modifying code 136
Semantic action 929
Semantic rule 929
Semaphores 263
Semiresident programs 1055
Sending a character to the printer via BIOS 1203
Sending a character to the printer via DOS 1203
Separate assembly 425
Separate compilation 425
Sequential logic 62
Serial chip input, testing for data available 1229
Serial data transmission 1199
Serial port I/O 1231
Serial port I/O addresses 1224
Serial port interrupt 1008
Serial port interrupt handlers 1239
Serial port loopback mode 1228
Serial port parity options 1231
Serial port, polled I/O 1236
Serial ports 1223
Set date (DOS) 718
Set interrupt vector call (DOS) 997
Set time (DOS) 718
SETcc instructions 281
SETL 565
Setting the autorepeat rate 293, 1168, 1358
Setting the baud rate on the serial chip 1225
Setting the number of serial port stop bits 1231
Setting the serial communications data size 1235
Setting the serial port baud rate 1231, 1234
Setting the serial port data size 1231
Seven segment decoder 61
Sharing interrupt vectors between ISRs 1010
SHELL.ASM 170
Shift instructions 243, 269, 270
Shift key status 293, 1168, 1358
Shift registers 64
SHL instruction 270, 271
SHLD instruction 270, 274
Short circuit evaluation 470
SHORT operator 392
SHR instruction 270, 273
SHRD instruction 270, 274
SI register 158
Side effects 602
Side effects in macros 419
Sign bit 23
Sign extension 25, 252, 268
Sign flag 244
Signed 23
Signed and unsigned numbers 23
Signed comparisons 282
Signed division 268
Signed integer variables 200
Significant digits 772
Simplification of boolean functions 52
Simulating keystrokes 1186
Sine 799
Single precision floating point format 775
Single step exception 1000
Sixteen-bit bus data access 89
Size of a processor 85
SIZE operator 392
SIZEOF operator 392
Skip routine (UCR Std Lib) 920
Sl_match2 routine (UCR Std Lib) 922
SNOBOL4 565
Software interrupts 995
Source index 820
Source index register 158
Spaghetti code 531
Spancset routine (UCR Std Lib) 914
Spanning strings 854
Spatial locality of reference 96
Special purpose registers 148
Square root 795
SR (set/reset) flip flop 62

SS register 155
STACK (SEGMENT operand) 373
Stack fault flag (FPU) 785
Stack frame 666
Stack-based parameters for procedures 574
Stalls 118
Standard entry code 582
Standard exit code 582
Start bits (serial chip) 1227
starting state 887
State machine 529
State machines 896
State variable 529
Static link 643
Statically allocated strings 831
Status register (FPU) 785
Status register (parallel port) 1201
STC instruction 302
STD instruction 302
STI instruction 302
Stop bits 1235
Stop bits (serial chip) 1227
Store instruction sequence (x86) 108
Stored program computer systems 101
Storing double words in byte addressable memory 87
Storing words in byte addressable memory 87
STOS 819, 828, 830
STOS instruction 284
StrBDel, StrBDelm string functions 846
Strcat, strcatl, strcatm strcatml functions 847
Strchr function 848
Strcmp, strcmpl functions 848
Strcpy, strcpyl functions 849
Strcspan, strcspanl functions 854
Strdel, strdelm functions 850
Strdup, strdupl functions 849
Stricmp, stricmpl functions 848
String assignment 832, 849
String comparison 834
String comparisons 848
String concatenation 844, 847
String constants 361
String deletion 850
String functions 835
String insertion functions 851
String instructions 243, 284, 819
String length 852
String length computation using SCAS 834
String primitives 819
String reversal 853
Strings 285, 819
Strins, strinsl, strinsm, strinsml functions 851
Strlen function 852
Strlwr, strlwrm functions 852
Strobe line (parallel port) 1200
Strongly type assembler 385
Strrev, strrevm functions 853
Strset, strsetm functions 853
Strspan, strspanl functions 854
Strstr, strstrl functions 855
Strtrim, strtrimm functions 855
STRUCT assembler directive 218
Structure initial values 220
Structure, accessing fields of... 219
Structures 218
Structures as structure fields 220
Strupr, struprm functions 852
Stuck parity 1228
Stuffing keys into the system keyboard buffer 1186
SUB instruction 259
Sub instruction sequence (x86) 108
Subroutine instance 642
Subroutines 289, 290
Substr (substring) 835
Substrings in patterns 925
Subtraction instructions 259
SUBTTL directive 424
Sum of minterms representation 49
Superscalar CPUs 123
Sword directive 384
SWORD pseudo-opcode 200
Symbol format 358
Symbol type 385
Symbol types 387
Symbol values 386
Symbolic addresses 358
Symbolic constants 360
Symbols 358
Synchronizing the FPU 801
Synthesizing a While loop 532
System bus 84
System clock 92
System clock frequency 93
System clock period 93
System timing 92

T

Table 493
Table generation 497
Tangent 799
Task switching with an FPU 802
TBYTE directive 384
TBYTE pseudo-opcode 202
TBYTE PTR operator 390
Temporal locality of reference 96
Temporary values in an expression 466
Term (boolean) 49
Terminate and stay resident programs 1025
Termination test (for loops) 532
Termination test for loops 535
Test for zero (floating point) 798
TEST instruction 279
Testing for an available key at the keyboard 293, 1168, 1358
Text constants 362

TEXTEQU directive 362
Theorems of boolean algebra 44
THIS operator 392
Three types of optimization 1315
Thunk 577
Timer interrupt 1007
Times (DOS) 718
Timing Delay Loops 544
TITLE directive 424
Toggle modifiers 1155
Trace exception 1000
Trace flag 245, 1186
Transient applications 1025
Transmitter empty flag (serial chip) 1230
Transmitting data between two computers 1209
Traps 995, 999
True (representation) 467
Truth maps 53
truth table 20
Truth tables 45
TSR 19
TSR identification 1035
TSR Installation 1035
TSR removal 1037
TSRs 1025
TTL logic levels 84
Turing machine 912
Two dimensional array model of segmentation 152
Two level caching system 98
Two's complement 16
Two's complement representation 23
Type ahead buffer 1008, 1158
Type ahead buffer (scan code insertion) 293, 1168, 1358
Type checking on BYTE values 199
Type conflicts 386
TYPE operator 392
Type operator 396
Type operators 392
TYPEDEF assembler directive 203
Types 385
Types of character strings 831

U

UCR Standard Library 333
UCR Standard Library floating point routines 777
UCR Standard Library string functions 845
Ultof (UCR Std Lib) 779
Unconditional JMP instructions 286
Underflow exception (FPU) 784
Unidirectional parallel port 1199
Unique boolean functions 46
Unit activation 642
Universal boolean function (NAND) 59
Universal boolean functions (NOR) 60
Unraveling loops 539
Unsigned comparisons 282
Unsigned division 267
unsigned multiplication 265
Unsigned numbers 23
Up code 1154
Upper case conversion 852
UTOA (UCR Std Lib) 341
Utof (UCR Std Lib) 779

V

Variable length parameters 592
Variable lifetime 641, 642
Variables 196, 384
Variables, byte 198
Variables, BYTE, initialized 200
Variables, double word 201
Variables, word 200
Virtual 8086 mode 149
VM (virtual machine) flag 149
Von Neumann, John 83

W

Wait states 95
While loop 532
Wildcard characters 883
Word access in byte addressable memory 87
Word directive 384
WORD pseudo-opcode 200
WORD PTR operator 390
Word ptr operator 472
Word strings 819
Word variables 200
Words 13, 15
Words stored at odd addresses 90
Wrappers (for nonreentrant code) 1033
Write control line 87
Writing data to the serial port 1232
Writing to memory 87
WTOA (UCR Std Lib) 341

X

x86 conditional jumps 106
x86 CPU registers 99
x86 instruction set 102
X86 mini-assembler 953
Xaccop routine (UCR Std Lib) 778
XADD instruction 256, 258
XLAT instruction 252, 255
XLIST (.XLIST) directive 425
XOR 467
XOR instruction 269
XOR operation 20, 21

Z

Zero divide exception (FPU) 784
Zero extension 252, 268
Zero flag 244
Zero terminated strings 831