The Essential Guide to SAS® Dates and Times

Second Edition

Derek P. Morgan

support.sas.com/bookstore

The correct bibliographic citation for this manual is as follows: Morgan, Derek P. 2014. *The Essential Guide to SAS® Dates and Times, Second Edition*. Cary, NC: SAS Institute Inc.

The Essential Guide to SAS® Dates and Times, Second Edition

Copyright © 2014, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-62959-066-0 (Hardcopy)
ISBN 978-1-62959-489-7 (EPUB)
ISBN 978-1-62959-490-3 (MOBI)
ISBN 978-1-62959-491-0 (PDF)

All rights reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a) and DFAR 227.7202-4 and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513-2414.

December 2014

SAS provides a complete selection of books and electronic products to help customers use SAS® software to its fullest potential. For more information about our offerings, visit **sas.com/store/books** or call 1-800-727-0025.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Contents

About This Book vii

Acknowledgments xi

Chapter 1: Introduction to Dates and Times in SAS 1

1.1 How Does It Work? (January 1, 1960, and Midnight as Zero) 1

1.2 Internal Representation 2

1.3 External Representation (Basic FORMAT Concepts) 2

1.4 Date and Time as Numeric Constants in SAS 3

1.5 Length and Numeric Requirements for Date, Time, and Datetime 5

1.6 General SAS Options for Dates 7

Chapter 2: Displaying SAS Date, Time, and Datetime Values as Dates and Times as We Know Them 9

2.1 How Do I Use a Format? 10

2.2 How Many Built-In Formats Are There for Dates and Times? 13

2.3 Date Formats, Justification, and ODS 13

2.4 Detailed Discussion of Each Format 14

- 2.4.1 Date Formats 14
- 2.4.2 Time Formats 37
- 2.4.3 Datetime Formats 41

2.5 Creating Custom Date Formats Using the VALUE Statement of PROC FORMAT 47

2.6 Creating Custom Date Formats Using the PICTURE Statement of PROC FORMAT 48

2.7 Creating Custom Formats Using PROC FCMP for Processing 52

2.8 The PUT() Function and Formats 55

Chapter 3: Converting Dates and Times into SAS Date, Time, and Datetime Values 57
3.1 Avoiding the Two-Digit Year Trap 57
3.2 Using Informats 59
3.3 The INFORMAT Statement 59
3.3.1 Using Informats with the INPUT Statement 60
3.3.2 Informats with the INPUT() Function 61
3.3.3 When the Informat Does Not Match the Data Being Read 62
3.4 Listing and Discussion of Informats 64
3.4.1 Date Informats 64
3.4.2 Time Informats 73
3.4.3 Datetime Informats 78
3.4.4 The "ANYDATE" Series of Informats 81
3.4.5 So Why Not Just Use the "ANYDATE" Series of Informats? 86
Chapter 4: ISO 8601 Dates, Times, Datetimes, Durations, and Functions 91
4.1 What Is ISO 8601? 91
4.2 ISO 8601 Formats 92
4.2.1 ISO Date Formats 93
4.2.2 ISO Time Formats 93
4.2.3 ISO Datetime Formats 99
4.3 ISO 8601 Informats 103
4.3.1 ISO Date Informats 104
4.3.2 ISO Time Informats 105
4.3.3 ISO Datetime Informats 108
4.4 Time Zone Functions 111
4.4.1 Introduction 111
4.4.2 The TIMEZONE= Option 111
4.4.3 List of Time Zone Functions 112

4.5 ISO 8601 Durations and Intervals 116
4.5.1 ISO Duration and Interval Representations 116
4.5.2 ISO 8601 Duration and Interval Formats 117
4.5.3 ISO 8601 Duration and Interval Informats 121
4.5.4 CALL IS8601_CONVERT 123
4.6 Conclusion 136
Chapter 5: Date and Time Functions 137
5.1 Current Date and Time Functions 137
5.2 Extracting Pieces from SAS Date, Time, and Datetime Values 138
5.3 Creating Dates, Times, and Datetimes from Numbers or Other Information 140
5.3.1 Introduction 140
5.3.2 List of Functions and Their Descriptions 140
5.4 Calculating Elapsed Time, and the HOLIDAY() Function 145
5.4.1 Calculating Elapsed Time with DATDIF() and YRDIF() 145
5.5 The Basics of SAS Intervals 149
5.5.1 The Interval Calculation Functions: INTCK() and INTNX() 151
5.6 Modifying SAS Intervals 159
5.7 Creating Your Own SAS Intervals 169
5.8 Interval Functions about Intervals 176
5.8.1 INTFIT(argument-1,argument-2,type) 177
5.8.2 INTFMT('interval','size') 178
5.8.3 INTGET(argument1,argument2,argument3) 179
5.8.4 INTSHIFT('interval') 180
5.8.5 INTTEST('interval') 181
5.9 Retail Calendar Intervals and Seasonality 181
5.9.1 Retail Calendar Intervals 181
5.9.2 Seasonality Functions 183
Chapter 6 Deeper into Dates and Times with SAS 185
6.1 Macro Variables and Dates 185
6.1.1 Automatic Macro Variables 185
6.1.2 Putting Dates into Titles 186
6.1.3 Using %SYSFUNC() to Create Dates, Times, and Datetimes in Macro Variables 187
6.1.4 Using Dates in Macros 189

6.2 Graphing Dates 194
Johnny's Savings Account 194
6.3 The Basics of PROC EXPAND 200
6.3.1 Capabilities of PROC EXPAND 200
6.3.2 Using PROC EXPAND to Convert to a Higher Frequency 202
6.3.3 Using PROC EXPAND to Convert to a Lower Frequency 203
6.3.4 Using PROC EXPAND to Interpolate Missing Values 205
6.3.5 The OBSERVED= Option for the CONVERT Statement in PROC EXPAND 206
6.4 International Date, Time, and Datetime Formats and Informats 212
6.4.1 "EUR" Formats and Informats 213
6.4.2 NLS Formats 214
6.5 NLS Date, Time, and Datetime Conversion Functions 221
6.6 Date Formats and Informats for Other Calendars 225
6.6.1 Hebrew Date Formats 226
6.6.2 Japanese and Taiwanese Date Formats 226
6.6.3 Japanese and Taiwanese Date Informats 226
6.7 Other Software and Their Dates (Excel, Oracle, DB2) 227
6.7.1 The SASDATEFMT= System Option 228
6.8 Conclusion 229
Appendix A: A Quick Reference Guide to SAS Date, Time, and Datetime Formats 231
Appendix B: A Quick Reference Guide to NLS Date, Time, and Datetime Formats 235
Appendix C: Troubleshooting Dates 101 239
Index 253

About This Book

Purpose

This book is designed to provide a detailed look at how the SAS date facility works, including an in-depth look at intervals and the interval functions, ISO 8601 date and datetime handling, and the NLS formats and informats. It is intended to serve as both a reference and a teaching tool. Ultimately, this book will allow the reader to become more confident in their daily work with dates, times, and datetimes in SAS.

Is This Book for You?

This book is aimed at beginning to intermediate SAS programmers, or those who work with ISO 8601 data, intervals, and/or reporting in multiple languages.

What's New in This Edition

This new edition includes updated information to reflect the changes in version 9 of SAS; an expanded discussion of intervals, including the ability to define your own intervals; a section on how SAS works with the ISO 8601 date standards; and a troubleshooting appendix for beginners.

Scope of This Book

This book does not cover the SAS/ETS product, except for an overview of the EXPAND procedure.

About the Examples

Software Used to Develop the Book's Content

SAS version 9.4 (TS level 1M0) was used to produce all the examples in this book.

Example Code and Data

Many of the examples used in this book have accompanying code and data.

You can access the example code and data for this book by linking to its author page at http://support.sas.com/publishing/authors. Select the name of the author. Then, look for the cover thumbnail of this book, and select Example Code and Data to display the SAS programs that are included in this book.

For an alphabetical listing of all books for which example code and data is available, see http://support.sas.com/bookcode. Select a title to display the book's example code.

If you are unable to access the code through the website, send e-mail to saspress@sas.com.

Output and Graphics Used in This Book

Tables in this book were generated using ODS RTF, while graphics were generated as PNG files directly in SAS using the GPLOT and SGPLOT procedures. Screen captures were used to show the VIEWTABLE displays.

Additional Help

Although this book illustrates many analyses regularly performed in businesses across industries, questions specific to your aims and issues may arise. To fully support you, SAS Institute and SAS Press offer you the following help resources:

- For questions about topics covered in this book, contact the author through SAS Press:
  - Send questions by email to saspress@sas.com; include the book title in your correspondence.
  - Submit feedback on the author's page at http://support.sas.com/author_feedback.
- For questions about topics in or beyond the scope of this book, post queries to the relevant SAS Support Communities at https://communities.sas.com/welcome.
- SAS Institute maintains a comprehensive website with up-to-date information. One page that is particularly useful to both the novice and the seasoned SAS user is its Knowledge Base. Search for relevant notes in the "Samples and SAS Notes" section of the Knowledge Base at http://support.sas.com/resources.
- Registered SAS users or their organizations can access SAS Customer Support at http://support.sas.com. Here you can pose specific questions to SAS Customer Support; under *Support*, click *Submit a Problem*. You will need to provide an email address to which replies can be sent, identify your organization, and provide a customer site number or license information. This information can be found in your SAS logs.

Meet the Author

Derek Morgan is a senior SAS programmer in the pharmaceutical industry who has been programming professionally in SAS for over 27 years. He spent 23 of those years at Washington University in St. Louis, where he received an A.B. in biology in 1985 and his first introduction to SAS as a student. During his career he has used SAS to create interactive data entry and management systems and to build and maintain research databases for analysis. In the late 1980s, he created a macro library to allow the use of nonproportional fonts in tables and listings on PostScript printers. He has taught introductory SAS programming and has presented many papers at local, regional, and national SAS Users Group conferences. Derek is married and has one son, and in his spare time he plays electric bass around the St. Louis area.

Learn more about this author by visiting his author page at http://support.sas.com/publishing/authors/morgan.html. There you can download free book excerpts, access example code and data, read the latest reviews, get updates, and more.

Keep in Touch

We look forward to hearing from you. We invite questions, comments, and concerns. If you want to contact us about a specific book, please include the book title in your correspondence.

Contact the Author through SAS Press

- By e-mail: saspress@sas.com
- Via the Web: http://support.sas.com/author_feedback

Purchase SAS Books

For a complete list of books available through SAS, visit sas.com/store/books.

- Phone: 1-800-727-0025
- E-mail: sasbook@sas.com

Subscribe to the SAS Training and Book Report

Receive up-to-date information about SAS training, certification, and publications via email by subscribing to the SAS Training & Book Report monthly eNewsletter. Read the archives and subscribe today at http://support.sas.com/community/newsletters/training!

Publish with SAS

SAS is recruiting authors! Are you interested in writing a book? Visit http://support.sas.com/saspress for more information.

Acknowledgments

A book is always a team effort, and I can honestly say that I have had a great one. Starting with the people at SAS Press whose efforts and patience on my behalf have resulted in this second edition, I would like to thank Julie Platt, Shelley Sessoms, and Stacey Hamilton but most especially my developmental editor, Stephenie Joyner, who kept me on task and moving forward. SAS guru Art Carpenter has been more than just an editor; he has been a mentor, and every time I run into him, I find myself knowing how to do something better than I did before. This second edition would not exist without his support. Another big thank you goes to Denise T. Jones, Stacy Suggs, and Robert Harris, all from SAS Press, who are responsible for producing this book from my manuscript.

Thanks to Andrew Karp for introducing me to the world of PROC EXPAND, and to Mike Forno of SAS Institute for answering my questions on it. I greatly appreciate the American Public Transportation Association (http://www.apta.com), which allowed me to use data compiled from its member transit agencies for the PROC EXPAND examples. The technical support group at SAS Institute deserves special mention; after 27 years of working as a computer programmer, I have come to the conclusion that you are, without a doubt, the best in the business. Thank you for all of your help throughout my SAS career.

I'd also like to thank the technical reviewers from SAS Institute—Richard Bell, Chris DeHart, Johnny Johnson, Rick Langston, and Kim Wilson—for their thoughtful comments and corrections. In addition, Paul Rowland deserves a big thank you for his willingness to provide a SAS user's perspective.

Ultimately, this project would not have been possible without the support of my family, past and present. To my son, Terec, thank you for understanding my distraction during Formula I season, and thank you for the photo. To my wife, Billie, thank you for letting me work on our weekends and a good number of our evenings, and for contributing your knowledge of Microsoft Word to help me write the manuscript.

Chapter 1: Introduction to Dates and Times in SAS

1.1 How Does It Work? (January 1, 1960, and Midnight as Zero)................. 1

1.2 Internal Representation 2

1.3 External Representation (Basic FORMAT Concepts).............................. 2

1.4 Date and Time as Numeric Constants in SAS 3

1.5 Length and Numeric Requirements for Date, Time, and Datetime 5

1.6 General SAS Options for Dates... 7

In the years that I've been working with SAS and teaching students how to use it, I find that two things consistently confuse those who are new to SAS. First is the default way that the DATA step works. Its implied DO-until-end-of-data generates many "How do I tell it how much data to read and when to stop?" questions. The second most confusing concept in SAS is that of how dates (and times) work within the software. I've seen many misuses of character strings masquerading as dates and/or times over the years, as well as unexpected results due to a failure to understand this fundamental part of SAS.

However, the way that SAS reads, stores, and displays dates and times is only the tip of the iceberg when it comes to the power and flexibility of SAS in handling this information. There is so much more than just having numbers represent date and time values. We'll start with the basics in the first three chapters, and then progress to some more advanced uses of those date and time values, taking advantage of many of the features available in SAS for that purpose.

1.1 How Does It Work? (January 1, 1960, and Midnight as Zero)

SAS counts dates, times, and datetime values separately. The date counter started at zero on January 1, 1960. Any day before January 1, 1960, is a negative number, and any day after that is a positive number. Every day at midnight, the date counter is increased by one. The time counter is measured in seconds and runs from zero (at midnight) to 86,400 (the number of seconds in a day), when it resets to zero. SAS calculates datetimes as the number of seconds since midnight, January 1, 1960. Why January 1, 1960? The founders of SAS wanted to use the approximate birth date of the IBM 370 system to represent the beginning of the modern computing era, and they chose January 1, 1960, as an easy-to-remember approximation.

In deciding whether to use a date, a time, or a datetime, you should consider how you are going to use it. Datetimes are always date and time combined; therefore, if you will not always have a time available for each date, you should strongly consider using separate date and time variables and then calculate a datetime variable from the two components when needed. Normally, attempting to create a datetime without both a date and a time will cause an error, and the result will be a missing value for the datetime. However, in specific circumstances, it is possible to create a datetime value from a date and a missing time (see Section 4.3.3, "ISO Datetime Informats," for an example). In these cases, the time will be set automatically to midnight (0 seconds of the given date). You may want your datetime value to be missing when there is a date but no time available. In these specific circumstances, it is important to keep track of date and time separately. Many programs that handle dates (such as databases and spreadsheets) maintain their dates and times as a numeric value relative to some fixed point in time, although the date that represents zero is different across each package, and packages may vary in how they keep track of time of day. Ultimately, this makes calculating durations easy, and working with dates and times stored in this fashion becomes a matter of addition, subtraction, multiplication, and division.

1.2 Internal Representation

SAS stores dates as integers, while the datetime and time counters are stored as real numbers to account for fractional seconds. The origin of the algorithm used for SAS date processing comes from a January 14, 1980, Computerworld article by Dr. Bhairav Joshi of SUNY-Geneseo. The earliest date that SAS can handle with this algorithm is January 1, 1582 (essentially the implementation date of the current Gregorian calendar system). The latest date is far enough into the future that at least five digits will be required to display the year.

Dates as stored by SAS don't do us much good in the real world. The statement "I was born on -242" won't mean much to anyone else. However, "I was born on May 4, 1959," can easily be translated into something that most people can understand, or it can be used as is. Fortunately, SAS has a number of built-in facilities to perform automatic translation between the internal numbers stored in SAS and dates and times and their representation as understood by the rest of the world. These built-in tools include formats and informats (introduced in Section 1.3 and covered extensively in Chapters 2, 3, and 4), date and time constants (Section 1.4), and functions (Chapter 5).

1.3 External Representation (Basic FORMAT Concepts)

Formats perform an automatic translation between the actual value and the value to be displayed. Formats display the date, time, and datetime values in a fashion that is much more easily understood. Formats do not change the values themselves; they are just a way to display the values in any output.

When you have dates or times and want to translate them into SAS date and time values, you will use informats. Although you will need a statement, procedure, or a function to actually create the

SAS values, informats describe what the data look like so that SAS can translate it correctly for storage. We will discuss formats and informats in detail in Chapters 2, 3, and 4 because there are dozens of them. Three of the most commonly used formats that work with SAS date, time, and datetime values are used in the following section.

1.4 Date and Time as Numeric Constants in SAS

We've talked about internal and external representation of dates and times. How do you put a specific date into a program as a constant? Formats only change the way the values are displayed in output, so you can't use them. Informats need a function or a SAS statement to translate the characters they are given, so you could use them, but then you would always need to use the INPUT() function to create a SAS date in a DATA step or PROC SQL. The INPUT function takes a series of characters that you give it and translates it using the informat that describes what the series of characters look like. That's very inefficient if you just want one specific date.

```
date = INPUT("04AUG2013",DATE9.);
```

Look at the program in Example 1.1 to see how date, time, and datetime constants are written into a SAS program. Take note of the quotation marks around the values for date, time, and datetime and the letters that follow each closing quote.

Example 1.1: Date Constants

```
DATA date_constants;

date = '04aug2013'd;  /* This is a date constant */
time = '07:15:00't;   /* This is a time constant */
datetime = '07aug1904:21:31:00'dt;  /* This is a datetime constant
*/
RUN;

TITLE "Unformatted Constants";
PROC PRINT DATA=date_constants;
VAR date time datetime;
RUN;

TITLE "Formatted Constants";
PROC PRINT DATA=date_constants;
VAR date time datetime;
FORMAT date worddate32. time timeampm9. datetime datetime19.;  /*
Format the constants */
RUN;
```

The quotes are used to create a literal value. You may use a pair of single or double quotes to specify the literal value. Dates have to be written as *ddmonyyyy*; times as *hh:mm:ss* (add a decimal point and more digits to represent fractional seconds if necessary); and datetimes as the date

ddmonyyyy, followed by a separator (frequently seen as a colon [:]) and then the time (*hh:mm:ss*). Aside from the correct formatting of these literal values, the most important part of a date/time/datetime constant is the letter that immediately follows the last quote. The letter "D" stands for date, "T" for time, and "DT" for datetime. Upper or lower case is valid. If you put one of these strings in quotes without the letter at the end, you will create a character variable, not a numeric variable with a date, time, or datetime value. The difference might not become apparent until you try to do something with the variable you created that involves a calculation. Don't forget your "D," "T," or "DT"! This example demonstrates how these constants are defined and then automatically converted to their equivalent SAS values.

The first PROC PRINT statement displays the date, time, and datetime values we created with our constants without formats, so we can see the values as they are stored in the data set.

Unformatted Constants		
date	**time**	**datetime**
19574	26100	-1748226540

The second PROC PRINT shows the effect of associating the variable DATE with the WORDDATE. format, the variable TIME with the TIMEAMPM. format, and the variable DATETIME with the DATETIME. format.

Formatted Constants		
date	**time**	**datetime**
August 4, 2013	7:15 AM	07AUG1904:21:31:00

Without the formats, you can see that the date constants we used to create the values stored in the data set are displayed as their actual SAS date, time, and datetime values. They don't make much sense to us until a format is associated with the variable.

What happens if you forget to put the "D," "T," or "DT" after your date constant? In Example 1.2, the "D," "T," and "DT" have been removed from the same date, time, and datetime in Example 1.1.

Example 1.2: Incorrect Date Constants

```
DATA bad_date_constants;

date = '04aug2013';  /* This is NOT a date constant */
time = '07:15:00';   /* This is NOT a time constant */
datetime = '07aug1904:21:31:00';  /* This is NOT a datetime constant
*/
RUN;

TITLE "Unformatted Constants";
PROC PRINT DATA=bad_date_constants;
```

```
VAR date time datetime;
RUN;
```

Now we print out the values without formats. While the problem may not be apparent at first glance, this result does not look like the unformatted SAS date, time, and datetime values in the previous example.

Unformatted constants		
date	**time**	**datetime**
04aug2013	07:15:00	07aug1904:21:31:00

Now let's try to add one day to the date, and a minute (60 seconds) to both the time and datetime. Here is a partial log of what happens when we try this with the code in Example 1.2.

```
12   DATA bad_date_constants;
13   date = '04aug2013' + 1;
14   time = '07:15:00' + 60;
15   datetime = '07aug1904:21:31:00' + 60;
16   RUN;

NOTE: Character values have been converted to numeric
      values at the places given by: (Line):(Column).
      13:8    14:8    15:12
NOTE: Invalid numeric data, '04aug2013' , at line 13 column 8.
NOTE: Invalid numeric data, '07:15:00' , at line 14 column 8.
NOTE: Invalid numeric data, '07aug1904:21:31:00' , at line 15 column
      12.
❶ date=. time=. datetime=. _ERROR_=1 _N_=1
```

The "invalid numeric data" note in the log tells you that you tried to use a character value to do something that requires a numeric value. The boldface last line tells you that you have missing values for all three variables, because you were trying to do math with a character value. Remember that SAS dates, times, and datetimes are always stored as numbers. When you see "invalid numeric data" where you intended to use a date constant, it is highly probable that your date constant is missing its identifying "D," "T," or "DT."

1.5 Length and Numeric Requirements for Date, Time, and Datetime

You can take advantage of the fact that dates are stored as integers to save space when you create variables to store them. Instead of using the default length of 8 for numeric variables, set the length of the numeric variables where you are storing dates to 4. This will safely store dates from January 1, 1582 (the earliest date SAS can handle), to October 23, 7701. A length of 5 is overkill, although that would extend the ending date another 534,773,760 days! A length of 3 will not accurately store

dates outside the range of January 1, 1960, and September 13, 1960. If you declare your date variables to be a length of 4, you will be able to store two dates in the space it would take to store one if you were using the SAS default length for numeric variables. This can save you a great deal of storage space in a large data warehouse.

Times may present a bit of a problem, because you may need to store fractional seconds. The rule is simple enough: If you want to store time values with fractional seconds, you *must* use a length of 8 to store them accurately. Otherwise, the length of 4 is long enough to store every possible time value from midnight to midnight down to the second. In these cases, not using the default length will allow you to store two times in the same amount of space as one.

Datetime values require more space, because a length of 4 will not store a datetime value with accuracy, regardless of whether you want fractional seconds. The number is just too big. As long as you are not storing fractional seconds, a length of 6 will store datetimes that accurately represent values from midnight on January 1, 1582, to 3:04:31 p.m. on April 9, 6315. Changing the range from the default of 8 to 6 for datetime values results in a 25 percent savings in space, which still may be significant depending on how much data you have. Of course, if you are going to maintain decimal places in your datetime values, you must use the default length of 8.

I have just provided the absolute minimum lengths required for accuracy. *DO NOT* attempt to save additional space by shrinking the variable lengths beyond 4, 6, or 8 as listed. You will lose precision, which could lead to unexpected results. Example 1.3 shows what can happen if you do not use enough bytes to store your date values. This example uses the value 19941, which represents the date of August 6, 2014, and it is in variables of lengths 3, 4, and 5.

Example 1.3: The Effect of LENGTH Statements on Dates

```
DATA date_length;
LENGTH len3 3 len4 4 len5 5;
len3 = 19941;
len4 = 19941;
len5 = 19941;
FORMAT len3 len4 len5 mmddyy10.;
RUN;
```

As the table below shows, when you try to store a date in fewer than 4 bytes, you do not get the correct value. Using a length of 4 to store your dates and times (without fractional seconds) is still a significant (50 percent) savings in the amount of storage required. You *will* create inaccuracies in your data if you try to save more than that. Saving additional space is not worth the risk of inaccurate data.

len3	len4	len5
08/05/2014	08/06/2014	08/06/2014

1.6 General SAS Options for Dates

Two options influence the default date and time stamp that SAS places on pages of output and the SAS log. The DATE/NODATE option causes the start date and time of the SAS job (or session) to appear on each page of the SAS log and SAS output. These values are obtained from the operating system clock and are displayed as 24-hour clock time, followed by the day of the week, month, day, and four-digit year. If you are running SAS interactively, then the date and time are printed only on the output, not the log. By default, the DATE system option is in effect when you start SAS. However, if you do not want this default display, then use the NODATE option. You probably don't want SAS to display its default date stamp if you are going to put your own date and/or time stamp in the title or in a footnote (see Chapter 6).

As mentioned in the previous paragraph, if the DATE option is enabled, SAS prints the date and time that the current SAS session started on each page. If you want a more exact date and time on those pages, you can use the DTRESET system option, which will cause SAS to retrieve the date and time from the operating system clock each time a page is written. That date and time will then be placed on the page instead of the time that the SAS job started. Since the time is displayed in hours and minutes, you will only see it change every minute. The DTRESET option can be useful in interactive applications or SAS programs that may have been running for days or weeks, where knowing when the output was generated is more important than knowing when the SAS session began. Since the DTRESET option affects the default SAS date and time stamp, it works only if the DATE option is enabled. When you use the NODATE option, using DTRESET will have no effect because you aren't using the SAS date and time stamp on your output.

Chapter 2: Displaying SAS Date, Time, and Datetime Values as Dates and Times as We Know Them

2.1 How Do I Use a Format? ... 10
2.2 How Many Built-In Formats Are There for Dates and Times? ... 13
2.3 Date Formats, Justification, and ODS ... 13
2.4 Detailed Discussion of Each Format ... 14
2.5 Creating Custom Date Formats Using the VALUE Statement of PROC FORMAT ... 47
2.6 Creating Custom Date Formats Using the PICTURE Statement of PROC FORMAT ... 48
2.7 Creating Custom Formats Using PROC FCMP for Processing ... 52
2.8 The PUT() Function and Formats ... 55

In SAS, date, time, and datetime values are stored as integers (unless you are storing fractional parts of seconds). They are all counted from a fixed reference point. SAS date values increment by 1 at midnight of each day, while SAS datetime values increment by 1 every second. SAS time values start at zero at midnight of each day, and also increment by 1 each second.

This scheme makes it easy to calculate durations in days and seconds, but it does not do much for figuring out what a given SAS date, time, or datetime value means in terms of how we talk about them. Therefore, SAS provides a facility that makes it easy to perform the translation from SAS into the common terminology of months, days, years, hours, and seconds. The translation is done through formats.

Formats are what SAS uses to control the way data values are displayed. They can also be used to group data values together for analysis. They are essential to dates and times in SAS because SAS does not store dates and times in an easily recognizable form, as we discussed in Chapter 1. SAS has many built-in formats to display dates, times, and datetime values. This chapter provides a detailed guide to all of the date, time, and datetime formats readily available in SAS. In addition, if any of these built-in formats don't fit your needs, you have the ability to create (and store for future use) your own formats. Creating your own formats is covered in Sections 2.5 and 2.6.

If you are looking for a quick reference, you can go to Appendix A, which lists all of the date, time, and datetime formats and provides a sample display using their default lengths. If the default does not give you what you want, Section 2.4 discusses each date, time, and datetime format in detail, including how to specify the length of the format and how that length affects the display.

2.1 How Do I Use a Format?

Formats are easy to use. You can permanently associate a format with a variable by using a FORMAT statement in a DATA step, as shown in Example 2.1.

Example 2.1: Permanently Associating a Format with a Variable

```
DATA test;
LENGTH date1 time1 4;
date1 = 19781;
time1 = 73000;
FORMAT date1 MMDDYY10. time1 TIMEAMPM11.;
RUN;
```

Example 2.1 creates a data set called TEST, which has two variables: date1 and time1. By using the FORMAT statement here, you have specified that whenever the values from this data set are displayed, the values stored in the variable date1 will always be displayed with the format MMDDYY10., and those stored in time1 will always be displayed using the TIMEAMPM11. format.

date1	time1
02/27/2014	8:16:40 PM

If you don't want to have your data values permanently associated with a format, then you can just apply the format when you are actually writing the values to your output. The same FORMAT statement is used, but the location has changed, from the DATA step to the PROC step. Example 2.2 illustrates this.

Example 2.2: Associating a Format with a Variable for the Duration of a Procedure

```
DATA test2;
LENGTH date2 time2 4;
date2 = 19781;
time2 = 73000;
RUN;

PROC PRINT DATA=test2;
FORMAT date2 DATE9. time2 TIMEAMPM11.;
RUN;
```

date2	time2
02/27/2014	8:16:40 PM

Although there is no format assigned to either date2 or time2 in the DATA step, you have told the PRINT procedure to write these values using the two formats listed, so there is no difference between the output from Example 2.1 and that from Example 2.2. Another handy thing about using the FORMAT statement with a SAS procedure is that if you use the FORMAT statement in a SAS procedure, it will override any format that has been permanently associated with the variables for the duration of that procedure. To illustrate, we'll take the data set TEST from Example 2.1. The variables date1 and time1 have been associated with the formats MMDDYY10. and TIMEAMPM11., respectively. What if your report needs the date printed out with the day of the week, along with the name of the month, day, and year, while the time needs to be seconds after midnight? The PROC PRINT step will look like this:

```
PROC PRINT DATA=test;
FORMAT date1 WEEKDATE37. time1;
RUN;
```

date1	time1
Thursday, February 27, 2014	73000

All SAS procedures will use the formats specified in the FORMAT statement that is part of the PROC step instead of the formats associated with the variable in the data set. Therefore, in the above example, date1 is printed using the WEEKDATE. format. What about time1? There's no format name given after the variable name in the FORMAT statement. This is how to tell SAS not to use any formats that might be associated with the variable. To remove a FORMAT from a

variable, make sure that no format names of any kind follow it anywhere in the FORMAT statement.

Example 2.3: Removing Associated Formats in a Procedure

```
DATA test3;
date3 = 19067;
time3 = 18479;
date4 = 18833;
time4 = 45187;
FORMAT date3 date4 DATE9. time3 time4 TIME5.;
RUN;

PROC PRINT DATA =test3 NOOBS;
FORMAT date3 date4 time3 time4;
RUN;
```

Data set test3 with all formats removed			
date3	**time3**	**date4**	**time4**
19067	18479	18833	45187

What happens if there are format names after the one that you want to remove? Look at the code segment that follows this paragraph. The goal is to display the variable date3 with the MMDDYY10. format, remove the format from the variable date4, and apply the TIMEAMPM11. format to the time variables time3 and time4. So the FORMAT statement has MMDDYY10. for date3, nothing for date4, and TIMEAMPM11. for time3 and time4, right?

```
PROC PRINT DATA =test3 NOOBS;
FORMAT date3 MMDDYY10. date4 time3 time4 TIMEAMPM11.;
RUN;
```

date3	time3	date4	time4
3/15/2012	5:07:59 AM	5:13:53 AM	12:33:07 PM

What happened? The variable date4 is displayed as a time, when you didn't put a format name after the variable name in the FORMAT statement. The answer is that you gave a list of three variables to SAS and told it to apply the TIMEAMPM11. format to them (see boldface code below):

```
FORMAT date3 MMDDYY10. date4 time3 time4 TIMEAMPM11.;
```

How do you fix this? You have to make sure that date4 is the last variable listed in the FORMAT statement.

```
PROC PRINT DATA =test3 NOOBS;
FORMAT date3 MMDDYY10. time3 time4 TIMEAMPM11. date4;
RUN;
```

date3	time3	date4	time4
3/15/2012	5:07:59 AM	18833	12:33:07 PM

2.2 How Many Built-In Formats Are There for Dates and Times?

SAS has more than 70 ready-to-use formats to display dates, times, and datetimes. We will discuss each one in detail in Section 2.4, but if you're looking for a quick reference guide, see Appendix A. SAS continues to develop formats and informats, so it is always a good idea to check the documentation that came with your release of SAS, or the online documentation at support.sas.com. All SAS formats have a common syntax structure, beginning with the format name, followed by an optional width specification, and ending with a period. The period is critical. It is what allows SAS to recognize the word as a format and not some other SAS keyword or text. The width specification varies with each format, and all formats have a default width that is used if there is no width specification given. The width specification is very important to dates because SAS will abbreviate the displayed value if you do not specify enough characters for the width, and the abbreviation that SAS uses might not give you the output that you want. The default width is noted in the description for each format in Section 2.4, and it is usually the width that will accommodate the longest value to be displayed. For example, the default width for the DOWNAME. (day-of-week) format is 9. That will accommodate the string "Wednesday", which is the longest English day-of-week name.

2.3 Date Formats, Justification, and ODS

Each date format has a default justification with respect to the width specification that you give it. Since numeric values are right-justified in SAS, most of the formats that are applied to date, time, or datetime formats are also right-justified, with a few exceptions (which will be clearly noted in the detailed explanations that follow). In ODS destinations other than LISTING, values are justified within a table column by SAS procedure default or by a user-defined ODS template. By default, SAS makes its columns wide enough to fit the widest item in a given column. Therefore, any leading spaces caused by specifying a width that is too wide to fit the formatted value won't show up in ODS output.

However, prior to ODS or version 9.3 in the LISTING destination, using a width specification that is wider than the output requires causes SAS to fill the empty spaces with blanks. For values that are right-justified, this might cause some of the output to shift to the right by a number of spaces. In

Example 2.4, we use the MONNAME. format, which displays the text corresponding to the month of the date, to illustrate.

Example 2.4: How Justification Works in the LISTING Destination

Format Name	Result	Comment
MONNAME9.	September	
MONNAME10.	□September	Leading space.
MONNAME11.	□□September	Two leading spaces.
MONNAME15.	□□□□□□September	Six leading spaces.

As you can see, making the width specification larger only adds leading spaces, and you could extend this all the way to the maximum width for the format.

Why should I worry about justification? I'm not using ODS LISTING, and I'm using SAS 9.3 or higher.

While it is true that justification is more of a concern in the traditional LISTING destination and only applicable to traditional column-based output, leading spaces can show up if you use the PUT() (or PUTN()) function to create character strings from SAS date, time, or datetime values. In cases such as these, the leading spaces are part of the output and as such might be displayed. You can use STRIP() or COMPRESS() to remove the leading spaces explicitly. If you are going to concatenate multiple items, use the CATX(), CATS(), or CATT() functions, all of which remove leading and trailing spaces of each item being concatenated.

If you do not specify column alignment in an ODS template or by using STYLE= directives, certain ODS destinations (such as RTF and PDF) will justify values within a column according to the justification of the format used in the column, without leading spaces.

2.4 Detailed Discussion of Each Format

This section gives a detailed explanation of all the current standard formats available for SAS date, time, and datetime values. In addition to the display that results from using a given format, the explanation includes information about the default width specification and its possible values, annotated examples of the display with varying width specifications, and usage notes. Date formats will be covered first, then time and datetime formats. Each subsection is arranged alphabetically.

2.4.1 Date Formats

A date format provides a set of instructions for how a SAS date is displayed so that it looks like a date in the way we normally express them. You can specify the width (number of characters) that

the translated text will occupy, but each format has its own default width specification, shown as *w* in this text. The default width specification is given in the description of each format. Some, but not all, of the date formats allow you to specify the character that separates each element of the date. You must not use a date format to translate datetime values. If you try to translate a datetime value with a date format, you will get incorrect output. (For an example, see Example 2.5.)

DATE*w*.

DATE*w*. writes dates as the numerical day of the month followed by the three-letter month abbreviation and the year, without any separating characters. It is right-justified within the field. *w* can be from 5 to 11, and the default width is 7. If you want to display four-digit years, use DATE9. or DATE11. DATE11. will display four-digit years with a hyphen between the day, month abbreviation, and year. The following table shows the result when the date value is 19715, which corresponds to December 23, 2013.

Format Name	Result	Comment
DATE.	23DEC13	Default width of 7.
DATE5.	23DEC	No room for year to be displayed.
DATE7.	23DEC13	Same as date.
DATE9.	23DEC2013	Four-digit year
DATE11.	23-DEC-2013	Hyphens as delimiters.

This format is analogous to the DTDATE. format, which displays datetime values in the same manner.

DAY*w*.

DAY*w*. writes the numerical day of the month, and it is right-justified within the field. *w* can be from 2 to 32, and the default width is 2. Specifying anything longer than 2 will only place more spaces in the field to the left of the number, so it is not necessary to specify more than 2. The following table shows the result when the date value is 16739, which corresponds to October 30, 2005.

Format Name	Result	Comment
DAY2.	30	

DDMMYY*w*.

DDMMYY*w*. writes dates as day/numerical month/year, where the slash (/) is the separator, and it is right-justified within the field. *w* can be from 2 to 10, and the default width is 8. If you specify a width from 2 to 5, the date will be truncated on the right, as SAS tries to fit as much of the day and

month as possible in the space allowed. If you use 6, no slashes will be printed. A width of 8 will use a two-digit year after the slashes. Use 10 to get a four-digit year with slashes. The following table shows the result when the date value is 19869, which corresponds to May 26, 2014.

Format Name	Result	Comment
DDMMYY5.	26/05	
DDMMYY6.	260514	
DDMMYY8.	26/05/14	
DDMMYY10.	26/05/2014	

DDMMYY*xw*.

DDMMYY*xw*. is similar to the DDMMYY. format. It is also right-justified. However, with this format, you can specify what character separates the day, numerical month, and year. The *x* in the format name represents the separator between the day, month, and year. The following table lists what *x* can be.

x	Character Displayed in Output	Comment
B	blank	
C	colon (:)	
D	dash (-)	
N	no separator	*w* is a maximum of 8, not 10.
P	period (.)	
S	slash (/)	Effectively the same as using the DDMMYY. format.

w can be from 2 to 10, with the default being 8. This works the same way as the DDMMYY. format with respect to what SAS fits in the space specified. Again, if you specify a width from 2 to 5, the date will be truncated on the right, as SAS tries to fit as much of the day and month as possible in the space allowed. If you use 6, no separator will be used. At 8, SAS will print a two-

digit year. Use 10 to get a four-digit year with your separator. The following table shows the result when the date value is 19398, which corresponds to February 9, 2013.

Format Name	Result	Comment
DDMMYYP5.	09.02	Only space for day and month.
DDMMYYB6.	090213	Not enough space for the blank separator.
DDMMYYD8.	09-02-13	Enough space for two-digit year with separators.
DDMMYYS8.	09/02/13	Enough space for two-digit year; same as using DDMMYY9. format.
DDMMYYC10.	09:02:2013	Colon as separator.

DOWNAME*w*.

DOWNAME*w*. writes the date as the name of the day of the week. It is right-justified, so if you give it too much space, there will be leading blanks. *w* can be from 1 to 32, and the default is 9. If you don't specify *w*, SAS will always print the entire name of the day. However, if you specify *w* less than 9, then SAS will truncate the name of the day to fit as necessary. The following table shows the result when the date value is 20280, which corresponds to Saturday, July 11, 2015.

Format Name	Result	Comment
DOWNAME3.	Sat	
DOWNAME6.	Saturd	
DOWNAME8.	Saturday	

JULDAY*w*.

JULDAY*w*. writes the date as the Julian day of the year, which is a value from 1 to 366. It is right-justified. *w* can be from 3 to 32, and the default is 3. Note that the minimum width of the format will cause leading spaces if the Julian date is less than 100. This would become obvious if you are creating a character string using the PUT() or PUTN() functions and do not remove leading blanks.

Format Name	Result	Comment
JULDAY3.	□□9	There are 2 leading spaces here because there are fewer than 3 digits in the value displayed. The date value used here is 374, which corresponds to January 9, 1961.

Format Name	Result	Comment
JULDAY3.	□76	There is a leading space here because there are fewer than 3 digits in the value displayed. The date value used here is 19068, which corresponds to March 16, 2012.
JULDAY3.	107	There is no leading space here because there are 3 digits in the value displayed. The date value used here is 19465, which corresponds to April 17, 2013.

JULIAN*w.*

JULIAN*w*. writes your date value as a Julian date, with the year preceding the Julian day. It is right-justified. *w* can be from 5 to 7, and the default is 5. If you specify a width of 5, the year portion of the Julian date is two digits long. If you specify a width of 7, the year portion is four digits long. The following table shows the result when the date value is 18514, which corresponds to September 9, 2010.

Format Name	Result	Comment
JULIAN5.	100252	
JULIAN7.	2010252	

MMDDYY*w.*

MMDDYY*w*. writes the date as numerical month/day/year, where a slash (/) is the separator. It is right-justified within the field. *w* can be from 2 to 10, and the default is 8. It is similar to the DDMMYY. format in that if you specify 2–5 for the width, the date will be truncated on the right, as SAS tries to fit as much of the day and month as possible in the space allowed. If you use 6, no slashes will be printed, but it will print a two-digit year. A width of 8 will put a two-digit year after the slashes. Use a width of 10 to get a four-digit year with slashes. The following table shows the result when the date value is 19655, which corresponds to October 24, 2013.

Format Name	Result	Comment
MMDDYY2.	10	Month only.
MMDDYY4.	1024	Month and day, no separator.
MMDDYY5.	10/24	Month and day with separating slash.
MMDDYY6.	102413	Month, day, and year, no separator.
MMDDYY8.	10/24/13	Year reduced to two-digit year to accommodate separators.
MMDDYY10.	10/24/2013	

MMDDYY*xw*.

MMDDYY*xw*. displays the date in the same way that the MMDDYY. format does, except that you can specify the separator. The *x* in the format name specifies the separator that you want to use according to the following table.

x	Character Displayed in Output	Comment
B	blank	
C	colon (:)	
D	dash (-)	
N	no separator	***w*** is a maximum of 8, not 10.
P	period (.)	
S	slash (/)	Effectively the same as using the MMDDYY. format.

The date will be right-justified within the width that you specify. *w* can be from 2 to 10, and the default is 8. If you specify 2–5, the date will be truncated on the right, as SAS tries to fit as much of the day and month as possible in the space allowed. If you use a width of 6, no separator will be used. At 7, SAS will print a two-digit year without a separator, and widths of 8 or 9 will put a two-digit year after the separator. Use 10 to get a four-digit year with separators. The following table shows the result when the date value is 19188, which corresponds to July, 14, 2012.

Format Name	Result	Comment
MMDDYYD5.	07-14	No room for year.
MMDDYYS6.	071412	No room for separators.
MMDDYYC8.	07:14:12	Colon as separator; still two-digit year.
MMDDYYP10.	07.14.2012	Period as separator.
MMDDYYB10.	07□14□2012	

MMYY*w*.

MMYY*w*. displays the zero-filled month number and year for the given date value, separated by the letter M. It is right-justified, and *w* can be from 5 to 32, with a default width of 7. When ***w*** is less than 7, a two-digit year is used. Otherwise, a full four-digit year is displayed. Since this format only needs a maximum of 7 characters, any width greater than 7 will just add leading spaces. The following table shows the result when the date value is 19756, which corresponds to February 2, 2014.

Format Name	Result	Comment
MMYY5.	02M14	
MMYY7.	02M2014	Four-digit year.

MMYY*xw.*

MMYY*xw*. displays the month number and year for a given date value in the same fashion that the MMYY. format does, except that you can specify the separator with *x*, according to the table below. Note that the blank is not valid with this format, while it is valid with the DDMMYYx. and MMDDYYx. formats.

x	Character Displayed in Output	Comment
C	colon (:)	
D	dash (-)	
N	no separator	***w*** can be from 4 to 32, with a default of 6.
P	period (.)	
S	slash (/)	

The displayed date value will be is right-justified, and *w* can be from 5 to 32, with a default width of 7. When *w* is specified as 5 or 6, a two-digit year is used, unless you have specified "N," for no separator. Without a separator, there is enough space to display the two digits of the month, and four digits for the year. If *w* is 7 or more, a full four-digit year is always displayed. Since this format will only display a maximum of 7 characters, any value greater than 7 will just add leading spaces. The following table shows the result when the date value is 19628, which corresponds to September 27, 2013.

Format Name	Result	Comment
MMYYD5.	09-13	Two-digit year.
MMYYN6.	092013	No separator, four-digit year.
MMYYS6.	□09/13	Two-digit year because of separator and leading space.
MMYYC7.	09:2013	Four-digit year.
MMYYP7.	09.2013	Four-digit year.

MONNAME*w*.

MONNAME*w*. displays the name of the month. It is right-justified, and *w* can be from 1 to 32, with a default of 9. Using a value greater than 9 will only add leading spaces. SAS will truncate the month name as necessary to fit in the width. The following table shows the result when the date value is 18336, which corresponds to September 15, 2010.

Format Name	Result	Comment
MONNAME3.	Sep	Specifying a *w* of 3 will display the three-letter month abbreviation.
MONNAME4.	Sept	
MONNAME9.	September	

MONTH*w*.

MONTH*w*. displays the number of the month of the year. It is right-justified, and *w* can be from 1 to 21, with a default of 2. Using a *w* of 1 will display the month number as a hexadecimal value (1 through C). The following table shows the result when the date value is 19341, which corresponds to December 14, 2012.

Format Name	Result	Comment
MONTH1.	C	***w*** of 1 always prints a single character, which is a hexadecimal digit.
MONTH2.	12	

MONYY*w*.

MONYY*w*. displays the three-letter month abbreviation, followed by the year without any separating characters. It is right-justified, and *w* can be from 5 to 7, with a default of 5. Specifying a width of 5 will give you a two-digit year. A width of 6 gives you a two-digit year and one leading space in the displayed date, while 7 gives you a four-digit year. It is analogous to the DTMONYY. format, which is used with datetime values. The following table shows the result when the date value is 19718, which corresponds to December 26, 2013.

Format Name	Result	Comment
MONYY5.	DEC13	
MONYY7.	DEC2013	

PDJULG*w.*

PDJULG*w*. writes a packed Julian date in hexadecimal format for IBM computers. Justification is not an issue, and *w* can range from 3 to 16. The default width is 4. The Julian date is written as follows: The four-digit Gregorian year is written in the first two bytes, and the three-digit integer that represents the day of the year is in the next one-and-a-half bytes. The last half-byte contains all binary 1s, which indicates the value is positive.

If the SAS date value being translated by this format is a date constant with a two-digit year, it will be affected by the YEARCUTOFF option. See the following SAS log.

```
246  OPTIONS YEARCUTOFF=1880;
247  DATA _NULL_;
248  date1 = "15JUN2004"d;
249  date2 = "15JUN04"d;  /* Affected by YEARCUTOFF option */
250  juldate1 = put(date1,PDJULG4.);
251  juldate2 = put(date2,PDJULG4.);
252  PUT juldate1= $HEX8.;
253  PUT juldate2= $HEX8.;
254  RUN;

juldate1=2004167F
juldate2=1904167F
```

PDJULI*w.*

PDJULI*w*. writes a packed Julian date in hexadecimal format for IBM computers. It only differs from the PDJULG. format in that it writes the century in the first byte as a two-digit integer, followed by two digits of the year in the second byte. The next one-and-a-half-bytes store the three-digit integer that corresponds to the day of the year, while the last half-byte is filled with hexadecimal 1s that indicate a positive number. As with the PDJULG. format, justification is not an issue, and the default width is 4, with a width range of 3 to 16.

The century and year are calculated by subtracting 1900 from the four-digit Gregorian year. A year value of 1980 gives a century/year value of 0080 (1980-1900=80), while 2015 gives 0115 (2015-1900=115). Be aware that this format will *not* produce correct results for years preceding 1900. The example below demonstrates.

```
OPTIONS YEARCUTOFF=2000;
DATA _NULL_;
  date1 = "15JUN1804"d;
  date2 = "15JUN1996"d;
  date3 = "15JUN96"d;
  juldate1 = PUT(date1,pdjuli4.);
  juldate2 = PUT(date2,pdjuli4.);
  juldate3 = PUT(date3,pdjuli4.);
  PUT juldate1= $hex8.;
  PUT juldate2= $hex8.;
```

```
  PUT juldate3= $hex8.;
  should_be_date1 = INPUT(juldate1,pdjuli4.);
  PUT should_be_date1= mmddyy10.;
  PUT should_be_date1= ;
RUN;
```

Here is the resulting output:

```
juldate1=009609DF
juldate2=0096167F
juldate3=0196167F
should_be_date1=04/12/1996
should_be_date1=13251
```

The value of date1 corresponds to a date in 1804, causing the PDJULI. representation of date1 (juldate1) to be incorrect. While it should be the same day of the year (167) as date2 and date3, you can see that the day is incorrectly written as 09D, while the century value is also incorrect, marked as 00 when, by the algorithm, it should be expressed as a negative number. This is verified by using the PDJULI. informat to read juldate1, which gives a result of April 12, 1996, when it should be June 15, 1804. (This is because there is no sign bit for julday1 to indicate that the value should be negative.) The difference between date2 and date3 is caused by the YEARCUTOFF option. The two-digit year 96 is translated as 2096, not 1996, because of the option's value.

QTR*w.*

QTR*w*. writes a date value as the quarter of the year. It is right-justified, and *w* can range from 1 to 32, with a default of 1. Since this format will only write 1 character, specifying a width greater than 1 will just add leading spaces. The following table shows the result when the date value is 18264, which corresponds to January 2, 2010.

Format Name	Result	Comment
QTR1.	1	

QTRR*w.*

QTRR*w*. also writes a date value as the quarter of the year, except that it displays the quarter as a Roman numeral. It is right-justified, and *w* can range from 3 to 32, with a default of 3. This format will write a maximum of 3 characters. A width specification greater than 3 will add leading spaces.

Format Name	Result	Comment
QTRR3.	III	The date value used is 17791 (September 16, 2008).

Format Name	Result	Comment
QTRR3.	□IV	With a date value of 17882 (December 16, 2008), there is 1 leading space.
QTRR5.	□□III	With a date value of 17791 (September 16, 2008), there are 2 leading spaces.

WEEKDATE*w.*

WEEKDATE*w*. writes date values as day-of-week name, month name, day, and year. It is right-justified, and *w* can range from 3 to 37. The default is 29, which is the maximum width of a date in this format. Specifying anything longer than 29 will cause leading spaces to be added. If the width specified is too small to display the complete day of the year and month, SAS will abbreviate. The following table shows the result when the date value is 18164, which corresponds to September 24, 2009.

Format Name	Result	Comment
WEEKDATE3.	Wed	Three-letter day-of-week abbreviation.
WEEKDATE9.	Wednesday	Will fit all day of week names. Leading spaces will be added for days other than "Wednesday."
WEEKDATE17.	Wed, Sep 24, 2009	Full date information, abbreviated.
WEEKDATE23.	Wednesday, Sep 24, 2009	Full day-of-week name and month name abbreviated.
WEEKDATE29.	Wednesday, September 24, 2009	

WEEKDATX*w.*

WEEKDATX*w*. writes date values as day-of-week name, day, month name, and year. It differs from the WEEKDATE. format in that the day of the month precedes the month name. It is right-justified, and *w* can range from 3 to 37. The default is 29, which is the maximum width of a date in this format. Specifying anything longer than 29 will cause leading spaces to be added. If the width specified is too small to display the complete day of the year and month, SAS will abbreviate. The following table shows the result when the date value is 19260, which corresponds to September 24, 2012.

Format Name	Result	Comment
WEEKDATX3.	Wed	Three-letter day-of-week abbreviation.
WEEKDATX9.	Wednesday	Will fit all day-of-week names. Leading spaces will be added for days other than "Wednesday."

Format Name	Result	Comment
WEEKDATX17.	Wed, 24 Sep, 2012	Full date information, abbreviated.
WEEKDATX23.	Wednesday, 24 Sep, 2012	Full day-of-week name and month name abbreviated.
WEEKDATX29.	Wednesday, 24 September, 2012	

WEEKDAY*w*.

WEEKDAY*w*. writes the date value as the number of the day of the week, where 1=Sunday, 2=Monday, and so on. It is right-justified, and *w* can be from 1 to 32. The default is 1. Since the maximum width of the display is always one character, specifying anything longer will just cause leading spaces to be added. The following table shows the result when the date value is 20569, which corresponds to Monday, April 25, 2016.

Format Name	Result	Comment
WEEKDAY1.	2	

WEEKU*w*.

WEEKU*w*. writes the date value as a week number in decimal format using the U algorithm. Unlike many other date, time, and datetime formats, it is *left-justified. w* can be from 1 to 200, and the default is 11. Specifying any value greater than 11 will display the same results as if *w* were 11. The U algorithm calculates weeks based on Sunday being the first day of the week, and the week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. The display that this format presents varies, based on the width specification. The following table shows the result when the date value is 20885, which corresponds to March 7, 2017, which is a Tuesday in the tenth week of the year.

Format Name	Result	Comment
WEEKU3.	W10	"W" indicates week; week number follows, with a leading zero if necessary.
WEEKU4.	W10	Same as WEEKU3. No leading spaces.
WEEKU5.	17W10	Two-digit year precedes week.
WEEKU6.	17W10	Same as WEEKU5.
WEEKU7.	17W1003	Two-digit year precedes week, week followed by the number of the day of the week.
WEEKU8.	17W1003	Same as WEEKU7.

Format Name	Result	Comment
WEEKU9.	2017W1003	Four-digit year precedes week; week number is followed by number of the day of the week.
WEEKU10.	2017W1003	Same as WEEKU9.
WEEKU11.	2017-W10-03	Separator added between year, week number, and number of the day of the week.
WEEKU12.	2017-W10-03	Same as WEEKU11.

WEEKV*w.*

WEEKV*w*. writes the date value as a week number in decimal format using the V algorithm, which is International Standards Organization (ISO) compliant. It is *left-justified* in the same fashion as the WEEKU. format. *w* can be from 1 to 200, and the default is 11. Specifying any value greater than 11 will display the same results as if *w* were 11. The V algorithm calculates weeks based on Monday being the first day of the week, and the week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary.

This algorithm defines the first week of the year as containing both January 4 and the first Thursday of the year. Therefore, if the first Monday of the year falls on January 2, 3, or 4, the preceding days of the calendar year are considered to be a part of week 53 of the previous calendar year. The following table shows the result when the date value is 19723, which corresponds to December 31, 2013. Note that although the date is in 2013, the algorithm used by this format places the date in the year 2014. Monday, December 30, 2013, is considered to be the first day of the first week for the year 2014.

Format Name	Result	Comment
WEEKV3.	W01	"W" indicates week; week number follows, with leading zero if necessary.
WEEKV4.	W01	Same as WEEKV3. No leading spaces.
WEEKV5.	14W01	Two-digit year precedes week.
WEEKV6.	14W01	Same as WEEKV5.
WEEKV7.	14W0102	Two-digit year precedes week; week followed by the number of the day of the week.
WEEKV8.	14W0102	Same as WEEKV7.
WEEKV9.	2014W0102	Four-digit year precedes week; week number is followed by number of the day of the week.

Format Name	Result	Comment
WEEKV10.	2014W0102	Same as WEEKV9.
WEEKV11.	2014-W01-02	Separator added between year, week number, and number of the day of the week.
WEEKV12.	2014-W01-02	Same as WEEKV11.

WEEKW*w*.

WEEKW*w*. writes the date value as a week number in decimal format using the W algorithm. As with the WEEKU. and WEEKV. formats, it is *left-justified*. *w* can be from 1 to 200, and the default is 11. Specifying any value greater than 11 will display the same results as if *w* were 11. The W algorithm calculates weeks based on Monday being the first day of the week without any other restriction. The week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. The display that this format presents varies based on the width specification. The following table shows the result when the date value is 19723, which corresponds to December 31, 2013 (the same date used in the V algorithm example). Note that the W algorithm assigns the date as the second day of the last week of the calendar year 2013.

Format Name	Result	Comment
WEEKW3.	W52	"W" indicates week; week number follows with leading zero if necessary.
WEEKW4.	W52	Same as WEEKW3. No leading spaces.
WEEKW5.	13W52	Two-digit year precedes week.
WEEKW6.	13W52	Same as WEEKW5.
WEEKW7.	13W5202	Two-digit year precedes week; week followed by number of day of week.
WEEKW8.	13W5202	Same as WEEKW7.
WEEKW9.	2013W5202	Four-digit year precedes week; week number is followed by number of the day of the week.
WEEKW10.	2013W5202	Same as WEEKW9.
WEEKW11.	2013-W52-02	Separator added between year, week number, and number of the day of the week.
WEEKW12.	2013-W52-02	Same as WEEKW11.

WORDDATE*w.*

WORDDATE*w*. displays the date value as name of month, day, and year. It is right-justified, and *w* can range from 3 to 32. The default is 18. If the width specified is less than 18, SAS will abbreviate the month name and add leading spaces as necessary, regardless of whether the specific date to be displayed will fit in the allocated space because of its value. This might have an impact if you are going to use the PUT() or PUTN() functions to create a character string using this format. The following table shows the result when the date value is 20328, which corresponds to August 28, 2015.

Format Name	Result	Comment
WORDDATE3.	Aug	
WORDDATE12.	Aug 28, 2015	
WORDDATE15.	□□□Aug 28, 2015	Three leading spaces.
WORDDATE18.	□□□August 28, 2015	Full month name, but only 15 characters—therefore, 3 leading spaces.
WORDDATE20.	□□□□□August 28, 2015	Five leading spaces.

WORDDATX*w.*

WORDDATX*w*. displays the date value as day, name of month, and year. It differs from the WORDDATE. format in that the day precedes the name of month. It is right-justified, and *w* can range from 3 to 32. The default is 18. If the width specified is less than 18, SAS will abbreviate the month name and add leading spaces as necessary, even if the date to be displayed will fit in the width specified. This might have an impact if you are going to use the PUT() or PUTN() functions to create a character string using this format. In the following table, you see that March is abbreviated for width specifications less than 18, even though there is room to print the entire date string. The table shows the result when the date value is 21259, which corresponds to March 16, 2018.

Format Name	Result	Comment
WORDDATX3.	Mar	
WORDDATX12.	□16 Mar 2018	Leading space.
WORDDATX14.	□□□16 Mar 2018	***w*** is less than 18, so the format uses the abbreviated month name and adds leading spaces even though the text displayed would fit in 13 characters.
WORDDATX16.	□□□□16 Mar 2018	***w*** is still less than 18, so "March" is still abbreviated, and more leading spaces are added.

Format Name	Result	Comment
WORDDATX18.	□□□□□16 March 2018	Printed with leading spaces because date string is only 13 characters long.

YEAR*w*.

YEAR*w*. displays the year for the given date value. It is right-justified, and *w* can be from 2 to 4, with a default width of 4. When *w* is specified as 2 or 3, a two-digit year is used. The following table shows the result when the date value is 18599, which corresponds to December 3, 2010.

Format Name	Result	Comment
YEAR2.	10	Two-digit year.
YEAR3.	□10	Two-digit year with a leading space.
YEAR4.	2010	Four-digit year.

YYMM*w*.

YYMM*w*. displays the year and month number for the given date value, separated by the letter M. It is right-justified, and *w* can be from 5 to 32, with a default width of 7. When *w* is specified as 5 or 6, a two-digit year is used. If *w* is 7 or more, a full four-digit year is displayed. Since this format can only display a maximum of 7 characters, anything more than 7 will just add leading spaces. The following table shows the result when the date value is 19517, which corresponds to June 8, 2013.

Format Name	Result	Comment
YYMM5.	13M06	Two-digit year.
YYMM6.	□13M06	Leading space.
YYMM7.	2013M06	Four-digit year.
YYMM8.	□2013M06	Leading space.

YYMM*xw*.

YYMM*xw*. displays the year and month number for a given date value in the same manner as the YYMM. format above, except that you can specify the separator with *x* according to the table below. Unlike the DDMMYYx., MMDDYYx., and the YYMMDDx. formats, a blank is not a valid separator with this format.

x	Character Displayed in Output	Comment
C	colon (:)	
D	dash (-)	
N	no separator	*w* can be from 4 to 32, with a default of 6.
P	period (.)	
S	slash (/)	

YYMM*xw*. is right-justified, and *w* can be from 5 to 32, with a default width of 7. When *w* is specified as 5 or 6, a two-digit year is used. If *w* is 7 or more, a full four-digit year is displayed. Specifying no separator with "N" will change the range of *w* from 4 to 32, and the default width becomes 6. Since this format can only display a maximum of 7 characters, anything more than 7 will just add leading spaces. The following table shows the result when the date value is 19237, which corresponds to September 1, 2012.

Format Name	Result	Comment
YYMMN4.	1209	No separator; minimum width is 4; two-digit year.
YYMMC5.	12:09	
YYMMD6.	□12-09	One leading space; two-digit year.
YYMMP7.	2012.09	Four-digit year.
YYMMS8.	□2012/09	Four-digit year; 1 leading space.

YYMMDD*w*.

YYMMDD*w*. is a variation on the DDMMYY. and MMDDYY. formats. It writes the date as a year, followed by the numerical month, and then the day, with a dash (-) as the separator. It is right-justified within the field. *w* can be from 2 to 10, and the default is 8. It is similar to the MMDDYY. format in that if you specify the width from 2 to 5, the date will be truncated on the right, as SAS tries to fit as much of the year and month as possible in the space allowed. If you use 6, no dashes will be printed. At 7, SAS will print a two-digit year without a dash, and 8 or 9 will produce a two-digit year before the first dash. Use a width of 10 to get a four-digit year with dashes. The

following table shows the result when the date value is 20031, which corresponds to November 4, 2014.

Format Name	Result	Comment
YYMMDD4.	1411	Two-digit year and month; not enough space for day.
YYMMDD5.	14-11	Two-digit year and month with dash separator; not enough space for day.
YYMMDD6.	141104	Two-digit year, month, and day; no separators.
YYMMDD7.	□141104	One leading space; no separators.
YYMMDD8.	14-11-04	Two-digit year.
YYMMDD9.	□14-11-04	Two-digit year with leading space.
YYMMDD10.	2014-11-04	Four-digit year.

YYMMDD*xw*.

YYMMDD*xw*. displays the date in the same way that the YYMMDD. format does, except that you can specify the separator. The *x* in the format name specifies the separator that you want to use according to the table below.

x	Character Displayed in Output	Comment
B	blank	
C	colon (:)	
D	dash (-)	Effectively the same as using the YYMMDD. format.
N	no separator	*w* is a maximum of 8, not 10.
P	period (.)	
S	slash (/)	

The date will be right-justified within the width that you specify. *w* can be from 2 to 10, and the default is 8. If you specify 2–5, the date will be truncated on the right, as SAS tries to fit as much of the day and month as possible in the space allowed. If you use 6, no separator will be used. At 7, SAS will print a two-digit year without a separator, and 8 or 9 will put a two-digit year before the first separator. The following table shows the result when the date value is 19500, which corresponds to May 22, 2013.

Format Name	Result	Comment
YYMMDDN4.	1305	Two-digit year and month; not enough space for day.
YYMMDDC5.	13:05	Two-digit year and month with colon separator; not enough space for day.
YYMMDDD6.	130522	Two-digit year, month, and day; no separators.
YYMMDDP7.	□130522	One leading space; not enough room for both separators.
YYMMDDB8.	13□05□22	Two-digit year with a blank separator.
YYMMDDN8.	20130522	Because there is no separator requested, a four-digit year is displayed.
YYMMDDS9.	□13/05/22	Enough room for separators, but only a two-digit year with a leading space.
YYMMDDD10.	2013-05-22	Four-digit year and dash separators; same as YYMMDD.

YYMON*w.*

YYMON*w*. writes dates as a two- or four-digit year followed by the three-letter month abbreviation. It is right-justified. *w* can be from 5 to 32, and the default is 7. Use a width of 7 to get a four-digit year. If *w* is less than 7, a two-digit year will be displayed. If *w* is larger than 7, leading spaces will be added. The following table shows the result when the date value is 20796, which corresponds to December 8, 2016.

Format Name	Result	Comment
YYMON5.	16DEC	Two-digit year.
YYMON6.	□16DEC	Two-digit year with 1 leading space.
YYMON7.	2016DEC	Four-digit year.
YYMON10.	□□□2016DEC	Four-digit year with 3 leading spaces.

YYQ*w.*

YYQ*w*. writes date values as a two- or four-digit year, followed by the letter Q, and a single-digit representing the quarter of the year. It is right-justified, and *w* can be from 4 to 32. The default width is 6. Use 6 to get a four-digit year, while a width of 4 or 5 will give you a two-digit year.

Specifying a width larger than 6 will only add leading spaces. The following table shows the result when the date value is 21384, which corresponds to July 19, 2018.

Format Name	Result	Comment
YYQ4.	18Q3	Two-digit year.
YYQ5.	□18Q3	Two-digit year with 1 leading space.
YYQ6.	2018Q3	Four-digit year.
YYQ8.	□□2018Q3	Four-digit year with 2 leading spaces.

YYQ*xw.*

YYQ*xw*. writes date values as a two- or four-digit year, followed by a separator that you specify, and a single-digit representing the quarter of the year. *x* is the letter that you use to indicate the separator according to the table below. Unlike the DDMMYY*x*., MMDDYY*x*., and the YYMMDD*x*. formats, a blank is not a valid separator with this format.

x	Character Displayed in Output	Comment
C	colon (:)	
D	dash (-)	
N	no separator	*w* can be from 3 to 32, with a default of 4. When *w* is 3 or 4, the year will be displayed with two digits.
P	period (.)	
S	slash (/)	

This format is right-justified, and *w* can be from 4 to 32. The default width is 6. Use a width of 6 to get a four-digit year, while 4 or 5 will give you a two-digit year. Specifying a width larger than 6 will add leading spaces. The following table shows the result when the date value is 19659, which corresponds to October 28, 2013.

Format Name	Result	Comment
YYQN3.	134	Two-digit year with no separator.
YYQC4.	13:4	Two-digit year.
YYQS5.	□13/4	Two-digit year with 1 leading space.

Format Name	Result	Comment
YYQP6.	2013.4	Four-digit year.
YYQD7.	□2013-4	Four-digit year with 1 leading space.

YYQR*w.*

YYQR*w*. writes date values as a two- or four-digit year, followed by the letter Q, and the quarter of the year is represented in Roman numerals. It is right-justified, and *w* can be from 6 to 32. The default width is 8. Use 8 to get a four-digit year, while 6 or 7 will give you a two-digit year. Specifying a width larger than 8 will add leading spaces.

The following table shows the result when the date value is 20312, which corresponds to August 12, 2015.

Format Name	Result	Comment
YYQR6.	15QIII	Two-digit year.
YYQR7.	□15QIII	Two-digit year with 1 leading space for the third quarter only; four-digit year displayed for the 1st, 2nd, and 4th quarters without any leading space.
YYQR8.	2015QIII	Four-digit year.
YYQR12.	□□□□2015QIII	Four-digit year with 4 leading spaces.

YYQR*xw.*

YYQR*xw*. writes date values as a two- or four-digit year, followed by a separator that you specify, and the quarter of the year is displayed as a Roman numeral. *x* is the letter that you use to indicate the separator according to the following table. This is another format that cannot use a blank as the separator.

x	Character Displayed in Output	Comment
C	colon (:)	
D	dash (-)	
N	no separator	*w* can be from 5 to 32, with a default of 7. When *w* is 5 or 6, the year will be displayed with two digits.
P	period (.)	

x	Character Displayed in Output	Comment
S	slash (/)	

This format is right-justified, and *w* can be from 6 to 32. The default width is 8. Use 8 to get a four-digit year, while 6 or 7 will display a two-digit year. Specifying a width larger than 8 will add leading spaces. The following table shows the result when the date value is 19545, which corresponds to July 6, 2013.

Format Name	Result	Comment
YYQRP6.	13.III	Two-digit year.
YYQRS7.	□13/III	Two-digit year with 1 leading space for the third quarter only; four-digit year displayed for the 1st, 2nd, and 4th quarters without any leading space.
YYQRN8.	□2013III	Four-digit year with 1 leading space and no separator.
YYQRC9.	□2013:III	Four-digit year with 1 leading space.
YYQRD10.	□□2013-III	Four-digit year with 2 leading spaces.

YYWEEKU*w*.

YYWEEKU*w*. writes the date value as a week number in decimal format using the U algorithm. Unlike many other date, time, and datetime formats, it is *left-justified. w* can be from 3 to 8, and the default is 7. The U algorithm calculates weeks based on Sunday being the first day of the week, and the week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. The display that this format presents varies based on the width specification. This format is similar to the WEEKU. format, except that it does not provide the numerical day of the week. The date value used in this example is 20041, which corresponds to November 14, 2014.

Format Name	Result	Comment
YYWEEKU3.	W45	"W" indicates week; week number follows; leading zero if necessary.
YYWEEKU4.	W45	Same as WEEKU3. No leading spaces because this format is left-justified.
YYWEEKU5.	14W45	Two-digit year precedes week.
YYWEEKU6.	14W45	Same as WEEKU5.

Format Name	Result	Comment
YYWEEKU7.	2014W45	Two-digit year precedes week; week followed by the number of the day of the week.
YYWEEKU8.	2014-W45	WEEKU7 with a dash as the delimiter.

YYWEEKV*w.*

YYWEEKV*w*. writes the date value as a week number in decimal format using the V algorithm, which is International Standards Organization (ISO) compliant. It is *left-justified* in the same fashion as the YYWEEKU. format. *w* can be from 3 to 8, and the default is 7. This format is similar to the WEEKV. format, except that it does not provide the numerical day of the week. The V algorithm calculates weeks based on Monday being the first day of the week, and the week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary.

This algorithm defines the first week of the year as containing both January 4 and the first Thursday of the year. Therefore, if the first Monday of the year falls on January 2, 3, or 4, the preceding days of the calendar year are considered to be a part of week 53 of the previous calendar year. The date value used in this example is 20455, which corresponds to Saturday, January 2, 2016. Note that although the date is in 2016, the algorithm used by this format places the date in week 53 of the year 2015. The first week of the year 2016 starts on Sunday, January 3, 2016, as that contains the first Thursday of the year and January 4, 2016.

Format Name	Result	Comment
YYWEEKV3.	W53	"W" indicates week; week number follows; leading zero if necessary.
YYWEEKV4.	W53	Same as WEEKU3. No leading spaces because this format is left-justified.
YYWEEKV5.	15W53	Two-digit year precedes week.
YYWEEKV6.	15W53	Same as WEEKU5.
YYWEEKV7.	2015W53	Two-digit year precedes week; week followed by the number of the day of the week.
YYWEEKV8.	2015-W53	WEEKU7 with a dash as the delimiter.

YYWEEKW*w.*

YYWEEKW*w*. writes the date value as a week number in decimal format using the W algorithm. As with the YYWEEKU. and YYWEEKV. formats, it is *left-justified*. *w* can be from 3 to 8, and the default is 7. This format is similar to the WEEKW. format, except that it does not provide the numerical day of the week. The W algorithm calculates weeks based on Monday being the first day of the week without any other restriction. The week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. The display that this format presents varies, based on the width specification. The following table shows the result when the date value 20455, which corresponds to Saturday, January 2, 2016 (the same date used in the YYWEEKV. algorithm example). Note that the W algorithm assigns the date as being in week zero of the calendar year 2016. Unlike the V algorithm, it does not consider the date as being a part of week 53 in 2015, but assigns it to week 0 in 2016. Week 1 is defined as the first full week in 2016.

Format Name	Result	Comment
YYWEEKW3.	W00	"W" indicates week; week number follows; leading zero if necessary.
YYWEEKW4.	W00	Same as WEEKU3. No leading spaces because this format is left-justified.
YYWEEKW5.	16W00	Two-digit year precedes week.
YYWEEKW6.	16W00	Same as WEEKU5.
YYWEEKW7.	2016W00	Two-digit year precedes week; week followed by the number of the day of the week.
YYWEEKW8.	2016-W00	WEEKU7 with a dash as the delimiter.

2.4.2 Time Formats

Time formats translate seconds into one of several different ways of displaying time. Only the TIMEAMPM. and TOD. formats are specific to clock values, displaying clock values from 12:00:00 a.m. to 11:59:59 p.m. All other formats will display hours greater than 23 when translating a value greater than or equal to 86400, which would be midnight of the following day. The display of minutes always ranges from 0 to 59, except when you are using the MMSS*w.d* format. The built-in SAS formats always display seconds from 0 to 59.

The width specification for time (and datetime) values is different from the one for date formats because it has to allow for decimal parts of seconds. Instead of *w*, time and datetime formats are specified as *w.d*, where *w* is the overall width of the entire format, and the *d* accounts for the number of digits to the right of the decimal point. *w* must be greater than (*d*+1) to account for the decimal point. As with date formats, each of these formats has its own default width specification, which is detailed in the description of the format.

HHMM*w.d*

HHMM*w.d* displays SAS time values as hours:minutes. It is right-justified and does not display a leading zero in front of the hours. *w* can be from 2 to 20, with a default of 5, while *d* indicates the number of decimal places to the right of the minutes. As previously noted, *w* must be greater than *d*+1, to account for the decimal point. It is different from the TIME*w.d* format in that it does not display seconds. If *d* is 0 or not present, SAS will round to the nearest minute. Otherwise, SAS will display the seconds in decimal minutes (seconds/60). Leading spaces will be added to the display based on the number of digits in the result. If the width specified is not long enough to accommodate the 3 spaces for the colon and the minutes, SAS will only display the hours and does not round the displayed value. The following table shows the result when the time value is 49794, which corresponds to 13 hours, 49 minutes, and 54 seconds.

Format Name	Result	Comment
HHMM2.	13	Hours only; no leading spaces because hour value is two digits.
HHMM3.	□13	Hours only; 2 leading spaces for single-digit hours; 1 leading space for double-digit hours. No leading spaces for three-digit hours. Not enough room for more than 999 hours.
HHMM5.	13:50	One leading space for single-digit hours, truncated to hours only if hours value is greater than 99 .
HHMM8.	□□□13:50	Three leading spaces
HHMM8.2	13:50.90	54 seconds = .9 minutes.

HOUR*w.d*

HOUR*w.d* displays SAS time values as hours and decimal fractions of hours. It is right-justified. *w* can be from 2 to 20, with a default of 2. *d* is the number of decimal places to the right of the hours, and *w* must be greater than *d*+1 to account for the decimal point. If you do not specify any decimal places, SAS rounds to the nearest hour. The following table shows the result when the time value is 53706, which corresponds to 14 hours, 55 minutes, and 6 seconds.

Format Name	Result	Comment
HOUR2.0	15	Rounded to the nearest hour.
HOUR4.2	14.9	55 minutes, 6 seconds, is .92 hours; does not leave enough space for a second decimal place.
HOUR6.2	□14.92	One leading space.

Format Name	Result	Comment
HOUR8.2	□□□14.92	Three leading spaces.

MMSS*w.d*

MMSS*w.d* displays SAS time values as minutes and seconds (mm:ss). It is right-justified. *w* can be from 2 to 20, with a default of 5. If you do not specify *w* large enough to fit minutes and seconds, SAS will round and display the minutes only. *d* will print decimal fractions (e.g., tenths or hundredths) of seconds. *w* must be greater than *d*+1 to account for the decimal point. The following table shows the result when the time value is 37269, which corresponds to the time 10:21:09 a.m. (10:21:09).

Format Name	Result	Comment
MMSS4.	□621	Leading space.
MMSS5.	□□621	Two leading spaces.
MMSS8.2	621:09.0	Only one decimal place for tenths of seconds, because there is not enough space to fit two. A *w* of 8 only leaves enough space for a *d* of 1 because of the decimal point.
MMSS9.2	621:09.00	Two decimal places for hundredths of seconds.

TIME*w.d*

TIME*w.d* displays SAS time values as hours:minutes:seconds. It is right-justified and does not print a leading zero in front of the hours. *w* can be from 2 to 20, with a default of 8, while *d* indicates the number of decimal places to the right of the seconds. *w* must be greater than *d*+1 to account for the decimal point. *d* will print decimal fractions (e.g., tenths or hundredths) of seconds.

This format is not restricted to a 24-hour day; if hours is greater than 24, then it will display the actual value of hours, which means that the default length of 8 may not be enough to hold the entire formatted time value. It is different from the HHMM. format in that it displays seconds as opposed to decimal fractions of minutes. If *d* is 0 or not present, SAS will round to the nearest second. The

time value used for the following table is 49794 seconds, the same value used for the HHMM*w.d* format. The time value corresponds to the time 1:49:54 p.m. (13:49:54).

Format Name	Result	Comment
TIME5.	13:49	
TIME6.	□13:49	Leading space.
TIME7.	□□13:49	Two leading spaces with seconds truncated. If this were a single-digit hour (i.e., 0 through 9), the entire time would be displayed without leading spaces because it would fit in 7 spaces.
TIME8.	13:49:54	Default will accommodate up to 99 hours.
TIME9.	□13:49:54	Leading space; may be necessary for hour values greater than 99.

TIMEAMPM*w.d*

TIMEAMPM*w.d* displays time in hours:minutes:seconds followed by a space and then AM or PM. It is right-justified. *w* can be from 2 to 20, and the default is 11. *w* must be greater than *d*+1, to account for the decimal point. Any time value greater than or equal to 86400 (midnight) will be displayed as the 12-hour clock time of the next day. This format does not print a leading zero. If you want the seconds to be printed, use at least 11 for the width. The following table shows the result when the time value is 11923, which corresponds to the time 3:18:43 a.m.

Format Name	Result	Comment
TIMEAMPM7.	□3:18 AM	Single-digit hour; 1 leading space; no seconds.
TIMEAMPM9.	□□3:18 AM	Two leading spaces; not enough room for seconds to be displayed.
TIMEAMPM11 .	□3:18:43 AM	Single-digit hour leaves 1 leading space.

TOD*w.d*

TOD*w.d* displays time in hours:minutes:seconds using the 24-hour clock. It is right-justified. *w* can be from 2 to 20, and the default is 11. *d* is the decimal fraction of seconds. *w* must be greater than *d*+1, to account for the decimal point. Any time value greater than or equal to 86400 (midnight of the next day) will be marked as the 24-hour clock time of the next day. This format does not print a leading zero. If you want the seconds to be displayed, use 8 for the width. Use at least 10 if you want decimal fractions of seconds shown. The following table shows the result when the time value is 75122, which corresponds to the time 8:52:02 p.m. (20:52:02):

Format Name	Result	Comment
TOD5.	20:52	
TOD8.	20:52:02	
TOD11.	□□□20:52:02	Three leading spaces; useful when using fractions of seconds.

2.4.3 Datetime Formats

Datetime formats translate SAS datetime values into one of several different formats. SAS datetime values are the number of seconds since midnight, January 1, 1960. You can use a format to display both the date and time. Again, you need to pay attention to the width specification in datetime formats because it allows for decimal fractions of seconds. Instead of *w*, time and datetime formats are specified as *w.d*, where *w* is the overall width of the entire format, and the *d* accounts for the number of digits to the right of the decimal point. *w* must be greater than *d*+1 to account for the decimal point. As with date formats, each of these formats has its own default width specification, which is detailed in the description of the format.

Starting with SAS version 9, there are also a number of "DT" formats that will allow you to display just the date or just the time from a datetime value, eliminating the need to use the DATEPART() or TIMEPART() functions for display purposes. Although these "DT" formats appear to give the same result as their corresponding date or time formats, the results you get will be very different should you use a datetime format on a date value and vice versa. Datetime formats translate *seconds* since midnight, January 1, 1960, while date formats translate *days* since January 1, 1960.

Example 2.5 shows what happens when you use date formats to interpret datetime values and vice versa. If you translate the SAS date value 19855 using a date format, you will get the correct value of May 12, 2014 (❶ and ❷). However, if you use a datetime format to translate the same value, you will get 5:30:55 a.m. on January 1, 1960, which corresponds to 19,855 seconds after midnight, January 1, 1960 (❸and ❹). In similar fashion, if you translate the value 1705587720 using a datetime format, you will get 2:22 p.m. on January 17, 2014 (❺ and ❻). This time, if you try to use a date format to translate this value, you will get a series of asterisks because the value is too far in the future for the SAS format algorithm to handle (❼ and ❽). Even if you try to get the month, day, and year of this value using the appropriate functions, it will not work.

Example 2.5: The Difference between Date and Datetime Values in Formats That Display Dates

```
DATA _NULL_;
date = 19855;
datetime = 1705587720;
PUT "MMDDYY10. representation of date=" date mmddyy10.; ❶
PUT "MONYY7. representation of date=" date monyy7.; ❷
PUT "DTMONYY7. representation of date=" date dtmonyy.; ❸
PUT "When value of date is used as a SAS *datetime* value, the date
represented is:" date datetime20.;❹
```

```
PUT "DATETIME20. representation of datetime=" datetime datetime20.;
❺
PUT "DTMONYY7. representation of datetime=" datetime dtmonyy7.; ❻
PUT "MONYY7. representation of datetime=" datetime monyy7.; ❼
PUT "When value of datetime is used as a SAS *date* value, the date
represented is:" datetime mmddyy10.; ❽
RUN;
```

And here is the result.

```
MMDDYY10. representation of date=05/12/2014 ❶
MONYY7. representation of date=MAY2014 ❷
DTMONYY7. representation of date=JAN60 ❸
When value of date is used as a SAS *datetime* value, the date
represented is:  01JAN1960:05:30:55 ❹
DATETIME20. representation of datetime=  17JAN2014:14:22:00 ❺
DTMONYY7. representation of datetime=JAN2014 ❻
MONYY7. representation of datetime=******* ❼
When value of datetime is used as a SAS *date* value, the date
represented is:********** ❽
```

With that in mind, here are the formats that are applicable to datetime values.

DATEAMPM*w.d*

DATEAMPM*w.d* displays datetime values as ddmonyy(yy):hh:mm:ss.ss xx, where *dd* is the day of the month, *mon* is the three-letter abbreviation for the month, and *yy(yy)* is the two- or four-digit year. A colon follows the date, and the time is represented by hh:mm:ss.ss, followed by a space and then AM or PM. It is right-justified. *w* can be from 7 to 40, and the default is 19. *d* is the decimal fraction of seconds and must be less than *w*–1, to account for the decimal point. *w* must be at least 13 to print AM or PM. If *w* is 10, 11, or 12, the time is displayed as a 24-hour clock. Also, if *w–d* is less than 17, the decimal values will be truncated to fit the specified field width. This format produces two-digit years for widths of 19 or less. The following table shows the result when the datetime value is 1746745319.6, which corresponds to the time 11:01:59.6 p.m. on May 8, 2015.

Note the rounding to the nearest second, which causes the displayed result to be rounded to the nearest minute when there isn't enough room for the decimal.

Format Name	Result	Comment
DATEAMPM7.	08MAY15	No room for time.
DATEAMPM12.	□□08MAY15:11	Two leading spaces; hours are given in 24-hour clock.
DATEAMPM18.	□□08MAY15:11:02 AM	Two leading spaces; not enough room for seconds.

Format Name	Result	Comment
DATEAMPM18.1	□□08MAY15:11:02 PM	Two leading spaces; not enough room for seconds.
DATEAMPM19.	08MAY15:11:02:00 PM	
DATEAMPM19.1	08MAY15:11:02:00 PM	Not enough room for decimal portion of seconds.
DATEAMPM21.	□□08MAY15:11:02:00 PM	No decimal specified. Result is rounded with leading spaces.
DATEAMPM21.1	08MAY15:11:01:59.6 PM	No leading spaces; entire result is displayed with no rounding.

DATETIME*w.d*

DATETIME*w.d* displays datetime values as ddmonyy(yy):hh:mm:ss.ss, where *dd* is the day of the month, *mon* is the three-letter month abbreviation, and *yy(yy)* is the two- or four-digit year. A colon follows the date, and the time is represented by hh:mm:ss.ss. It is similar to the DATEAMPM*w.d* format, except that it uses the 24-hour clock and therefore does not display AM or PM. It is right-justified. *w* can be from 7 to 40, and the default is 19, which will provide enough room for seconds. Use a length of 15 if you only want hours and minutes to be displayed along with the date. *d* is the decimal fraction of seconds and must be less than *w*–1 to account for the decimal point. If *w–d* is less than 17, the decimal values will be truncated to fit the specified field width. If *w–d* is less than 19, this format produces two-digit years. The following table shows the result when the datetime value is 1698258126.84, which corresponds to the time 6:22:07 p.m. on October 24, 2013.

Format Name	Result	Comment
DATETIME16.	24OCT13:18:22:07	
DATETIME18.	□□24OCT13:18:22:07	Two leading spaces.
DATETIME18.1	24OCT13:18:22:06.8	One decimal place.
DATETIME19.	24OCT2013:18:22:07	Four-digit year; not enough space requested for decimal point and decimal place.
DATETIME19.1	□24OCT13:18:22:06.8	*w–d* =18, so the year is shown as a two-digit year with 1 leading space.
DATETIME20.	□□24OCT2013:18:22:07	Two leading spaces.
DATETIME20.1	24OCT2013:18:22:06.8	*w–d* = 19, so the year is shown as a four-digit year.
DATETIME21.	□□□24OCT2013:18:22:07	Three leading spaces.

Format Name	Result	Comment
DATETIME21.2	24OCT2013:18:22:06.84	Four-digit year; decimal seconds to 2 places.

DTDATE*w*.

DTDATE*w*. displays datetime values as the numerical day of the month, followed by the three-letter month abbreviation and the year without any separating characters. It is right-justified within the field. *w* can be from 5 to 9, and the default width is 7. If you want to display four-digit years, use DTDATE9. The output is identical to the output using the DATE. format. The difference is that this format will only work correctly with datetime values, while the DATE. format only works correctly with date values. The following table shows the result when the datetime value is 1729519835.7, which corresponds to the time 2:10:36 p.m. on October 21, 2014.

Format Name	Result	Comment
DTDATE5.	21OCT	
DTDATE6.	□21OCT	Leading space; no year.
DTDATE7.	21OCT14	
DTDATE8.	□21OCT14	Leading space.
DTDATE9.	21OCT2014	Four-digit year.

DTMONYY*w*.

DTMONYY*w*. displays the date from a datetime value as the three-letter month abbreviation followed immediately by the year. There are no separating characters. It is right-justified, and *w* can be from 5 to 7, with a default of 5. Specifying 5 or 6 will give you a two-digit year, while 7 will give you a four-digit year. Although this format appears to produce the same MMMyy(yy) result as the MONYY. format, the DTMONYY. format only works with datetime values, while the MONYY. format only works with date values. The following table shows the result when the datetime value is 1727036339.1, which corresponds to the time 8:18:59 p.m. on September 22, 2014.

Format Name	Result	Comment
DTMONYY5.	SEP14	
DTMONYY6.	□SEP14	Leading space.
DTMONYY7.	SEP2014	Four-digit year.

DTWKDATX*w.*

DTWKDATX*w*. writes datetime values as day of week name, day, month name, and year. It differs from the WEEKDATX. format in that it works on datetime values, not date values. It is right-justified, and *w* can range from 3 to 37. The default is 29, which is the maximum width of a date in this format. Specifying anything longer than 29 will cause leading spaces to be added. If the width specified is too small to display the complete day of the year and month, SAS will abbreviate. It will first abbreviate the month and then the day of the week as necessary. The following table shows the result when the datetime value is 1645971726, which corresponds to the time 2:22:06 p.m. on February 27, 2012.

Format Name	Result	Comment
DTWKDATX3.	Mon	
DTWKDATX9.	□□□Monday	Full name of day with leading spaces. Not enough room to display the rest of the value.
DTWKDATX15.	□Mon, 27 Feb 12	Leading space.
DTWKDATX23.	□□□□Monday, 27 Feb 2012	Full name of day; month abbreviation; four-digit year.
DTWKDATX29.	□□□□□Monday, 27 February 2012	Leading spaces in example, but will fit any date.

DTYEAR*w.*

DTYEAR*w*. displays the year for the given datetime value. The DTYEAR. format is identical in result to the YEAR. format, but it is used with datetime values instead of date values. It is right-justified, and *w* can be from 2 to 4, with a default width of 4. When *w* is specified as 2 or 3, a two-digit year is used. The following table shows the result when the datetime value is 1716310050.2, which corresponds to the time 4:47:30 p.m. on May 21, 2014.

Format Name	Result	Comment
DTYEAR2.	14	Two-digit year.
DTYEAR3.	□14	Two-digit year with a leading space.
DTYEAR4.	2014	Four-digit year.

DTYYQC*w*.

DTYYQC*w*. writes datetime values as a two- or four-digit year, followed by a colon, and a single-digit representing the quarter of the year. It is right-justified, and *w* can be from 4 to 6. The default width is 4. Use a width of 6 to get a four-digit year. Use 4 or 5 to get a two-digit year. This gives you the same result with datetime values that using the YYQC. format would yield with a date value. The following table shows the result when the datetime value is 1662064659.5, which corresponds to the time 8:37:40 p.m. on August 31, 2012.

Format Name	Result	Comment
DTYYQC4.	12:3	Two-digit year.
DTYYQC5.	□12:3	Leading space; two-digit year.
DTYYQC6.	2012:3	Four-digit year.

MDYAMPM*w*.

MDYAMPM*w*. writes datetime values in the following format: mm/dd/yy(yy) □hh:mm□AM/PM. There are no leading zeroes in either the date or the time portions of this format. It is right-justified, and *w* can be from 8 to 40, with a default width of 19. The following table shows the result when the datetime value is 1716089122, which corresponds to the time 3:25:22 a.m. on May 19, 2014. Note that in the following example there are two spaces between the date and the time because the hour is only a single digit.

Format Name	Result	Comment
MDYAMPM8.	□5/19/14	In SAS versions earlier than 9.4, the date is represented as 0/0/0 for lengths less than 17.
MDYAMPM15.	5/19/14	
MDYAMPM16.	5/19/14	
MDYAMPM17.	5/19/14 3:25 AM	
MDYAMPM18.	5/19/14 3:25 AM	
MDYAMPM19.	5/19/2014 3:25 AM	
MDYAMPM21.	5/19/2014 3:25 AM	
MDYAMPM24.	5/19/2014 3:25 AM	

2.5 Creating Custom Date Formats Using the VALUE Statement of PROC FORMAT

In addition to the date and time formats supplied with SAS, you can create your own custom formats with the FORMAT procedure. With dates and times, you can modify the default display of an existing SAS format or create your own using the VALUE or the PICTURE statement. Below I show how to modify the default display of an existing SAS format using the VALUE statement.

Example 2.6: Creating Your Own Date Format with the VALUE Statement in PROC FORMAT

An access control company wants a report of the people whose security cards have expired as of January 1, 2014, and they have the expiration date for each card. Instead of having to read the report and determine which dates are prior to the cutoff, they want to display any date prior to January 1, 2014, as "EXPIRED," and dates after that using the MMDDYY10. format. Set the value to "EXPIRED" for dates prior to December 31, 2013, using that as the upper limit, and the special value "LOW" as the bottom of the range. For the remaining values, use the lower limit of January 1, 2014, and the special value "HIGH" as the upper limit, and indicate the SAS format to be applied to this range in brackets ❶. If you do not enclose the SAS format in brackets, you will get an error in the FORMAT procedure.

```
PROC FORMAT;
VALUE exp
LOW-'31DEC2013'd= "Expired"
'01JAN2014'd - HIGH=[ MMDDYY10.] ;❶
RUN;

PROC PRINT DATA=access;
ID card_num;
VAR exp_date exp_date_raw;
FORMAT exp_date exp. exp_date_raw DATE9.;
RUN;
```

card_num	exp_date	exp_date_raw
84485598	11/14/2015	14NOV2015
16205371	11/27/2014	27NOV2014
63656754	01/14/2014	14JAN2014
10270040	Expired	01APR2013
94822015	Expired	04JUN2013
27800904	Expired	23OCT2013
97189418	08/14/2014	14AUG2014
70815194	03/14/2016	14MAR2016
50465401	Expired	26MAY2013

card_num	exp_date	exp_date_raw
43034970	09/28/2014	28SEP2014

Example 2.7: Creating Your Own Time Format with the VALUE Statement in PROC FORMAT

In order to be able to drive the next stage in a road race, drivers must finish a previous stage in ten minutes or less, and the results are posted. This example shows that you can customize time and datetime formats as well as date formats. To customize datetime formats, you would specify a datetime format instead of a date or time format.

```
PROC FORMAT;
VALUE qualify
LOW-'00:10:00't=[MMSS5.];
'00:10:00't <- HIGH = 'Did Not Qualify';
RUN;

PROC PRINT DATA=racers;
ID name;
VAR time;
FORMAT time qualify.;
RUN;
```

name	time
Bork	Did Not Qualify
Bova	Did Not Qualify
Brantley	08:31
Brickowski	08:59
Burkhart	07:10
Burroughs	08:05
Butler	)id Not Qualify

2.6 Creating Custom Date Formats Using the PICTURE Statement of PROC FORMAT

To create your own date and time formats with the PICTURE statement in the FORMAT procedure, you need to use the DATATYPE= option. DATATYPE can take the values of "DATE," "TIME," or "DATETIME" to indicate the type of value you are formatting. You then need to define your display by using one or more of the picture format date directives shown in the list below. These directives *are* case-sensitive.

Table 2.1: Picture Format Date Directives

Date Directive	Description
%a	Locale's abbreviated weekday name. Locale is defined by the LOCALE= system option.
%A	Locale's full weekday name. Locale is defined by the LOCALE= system option.
%b	Locale's abbreviated month name. Locale is defined by the LOCALE= system option.
%B	Locale's full month name. Locale is defined by the LOCALE= system option.
%d	Day of the month as a decimal number (1–31), with no leading zero. Put a zero between the percent sign and the "d" to have a leading zero in the display.
%H	Hour (24-hour clock) as a decimal number (0–23), with no leading zero. Put a zero between the percent sign and the "H" to have a leading zero in the display.
%I	Hour (12-hour clock) as a decimal number (1–12), with no leading zero. Put a zero between the percent sign and the "I" to have a leading zero in the display.
%j	Day of the year as a decimal number (1–366), with no leading zero. Put a zero between the percent sign and the "j" to have a leading zero in the display.
%m	Month as a decimal number (1–12), with no leading zero. Put a zero between the percent sign and the "m" to have a leading zero in the display.
%M	Minute as a decimal number (0–59), with no leading zero. Put a zero between the percent sign and the "M" to have a leading zero in the display.
%p	Locale's equivalent of either a.m. or p.m. Locale is defined by the LOCALE= system option.
%S	Second as a decimal number (0–59), with no leading zero. Put a zero between the percent sign and the "S" to have a leading zero in the display.
%U	Week number of the year (Sunday as the first day of the week) as a decimal number (0–53), with no leading zero. Put a zero between the percent sign and the "U" to have a leading zero in the display.
%w	Weekday as a decimal number, where 1 is Sunday and Saturday is 7.
%y	Year without century as a decimal number (0–99), with no leading zero. Put a zero between the percent sign and the "y" to have a leading zero in the display.
%Y	Year with century as a decimal number (four-digit year).
%%	The percent character (%).

Note that if you are going to use these directives in a picture format, you will probably want to add the following line of code or some variant in the PICTURE statement to handle the display of missing values.

```
. - .z = "No Date Given"
```

Example 2.8 creates a format similar to the WORDDATE. format, except that leading zeroes are a part of the date display. Pay special attention to how the picture is created in line 4. When you are using any of the date directives from the table above, you *must* enclose your picture string in *single* quotes. If you use double quotes, SAS will try to interpret the directives as macro calls because of the percent sign (%). The PROC FORMAT step will execute without error, and you will not see any warnings in the SAS log until the format is used. The format will not work if you have a macro that has the same name as the format you created.

Example 2.8: Creating a Picture Format for Dates Using Date Directives

```
1 PROC FORMAT (DEFAULT=21);
2 PICTURE ZWDATE
3 . - .z = "No Date Given"
4 LOW - HIGH = '%B %0d, %Y' (DATATYPE=DATE);
5 RUN;

6 PROC PRINT DATA=pictest LABEL;
7 VAR date date2;
8 FORMAT date worddate. date2 zwdate21.;
9 LABEL date = "Date Using WORDDATE."
10 date2 = "Date using ZWDATE.";
11 RUN;
```

Line 3 defines the display for missing values. Without it, you will see the SAS missing value symbol: either a period (.) or a special missing value. Line 4 gives the picture of how the date is to be displayed. The zero preceding the day directive (%0d) causes the leading zero to be printed as a part of the date. The length of the string with the date directives determines the default width of the format. In this example, the default width is 10, but in order to make sure that all the dates print correctly, the FORMAT statement in line 8 sets the format width to 21. What follows is the resulting output.

Date using WORDDATE.	Date using custom PICTURE format ZWDATE.
July 8, 2012	July 08, 2012
April 17, 2014	April 17, 2014
October 30, 2013	October 30, 2013
.	No Date Given
May 14, 2012	May 14, 2012
February 2, 2013	February 02, 2013
.	No Date Given
November 26, 2014	November 26, 2014

Example 2.9: Using Date Directives with Text

This example defines a format that will display text along with the date and time. It provides a standardized date and time stamp that can be added as a title or footnote to reports (for examples of how to do this, see Sections 6.1.2 and section 6.1.3. In line 2 of the code below, the DEFAULT= option is used to specify a default length for the format. Without that option, this format would have a default length of 35, which is the number of characters between the quotes. However, there is no way that the default width can successfully print the entire date and time stamp for all cases, because full month names are always longer than two characters, which is all that the format directive "%B" makes room for. Similarly, the four-digit year will take up more than the two characters accounted for by the %Y directive. The code below uses the format width of 35 to illustrate what would happen without the DEFAULT= option.

In line 3, if the datetime stamp is missing, the datetime stamp displays the word "Missing." The picture for all other values is defined in line 4; "LOW-HIGH" specifies the range. The picture description begins with text, and the date is represented by the full month name (%B), followed by the numeric day (%d without a leading zero), a comma, and the four-digit year **(**%Y). The time follows the word "at" and is represented as a 12-hour clock time without seconds (%I:%0M), followed by AM or PM (%p). The DATATYPE= option tells the format that it will be receiving datetime values to translate.

SAS Code

```
1 PROC FORMAT;
2 PICTURE rptdate (DEFAULT=43)
3 . - .Z = 'Missing'
4 LOW-HIGH = 'Generated on %B %d, %Y at %I:%0M %p'
(DATATYPE=DATETIME);
5 RUN;
6
7 DATA _NULL_;
8 rpt_date = "15dec2012:22:25:00"dt;
9 PUT 'SAS datetime value = ' rpt_date;
10 PUT 'Formatted with DATETIME. = ' rpt_date datetime.;
11 PUT 'Using custom format at width of 35 =' rpt_date rptdate35.;
12 PUT 'Using custom format at default width of 43 = ' rpt_date
rptdate.;
12 RUN;
```

The Result

```
SAS datetime value = 1671229500
Formatted with DATETIME. = 15DEC12:22:25:00
Using custom format at width of 35 =Generated on December 15, 2012
at 1 ❶
Using custom format at default width of 43 =  Generated on December
15, 2012 at 10:25 PM
```

As you can see, without the corrected default width, the resulting output is truncated at 35 characters.

2.7 Creating Custom Formats Using PROC FCMP for Processing

There is another possibility for custom formats beyond the functionality of the VALUE and PICTURE statements. You can actually process data using PROC FCMP to write your own functions and then display the result of that function as a formatted value. This can be helpful for handling dirty date data (missing values for month, day, or year) or imputing values for missing date components. While the specifics of PROC FCMP are beyond the scope of this book, the following section will demonstrate some of the capability.

Example 2.10: Using PROC FCMP to Impute Dates with Annotation for Imputed Dates

This example defines a format that will take a date, and if any component is missing, it will impute a value for display on the report along with an indicator of which component is missing. The advantage of doing this with a format is that the original value in the data set is untouched, allowing for investigation and later correction. The OUTLIB= option (❶) on the PROC FCMP statement defines the location where the compiled function is to be stored. The FUNCTION statement (❷) names the function and describes the input parameter (cdate; the $ indicates that it is a character value), and the second dollar sign indicates that the returned value will also be character.

PROC FCMP Code to Create the MAKEDATE() Function for Imputation

```
ODS ESCAPECHAR ='~';

PROC FCMP OUTLIB=control.functions.dates;  ❶

 FUNCTION makedate(cdate $)$;  ❷
   LENGTH dtstr $42 supr  $16;
   supr=' '; /* Initialize superscript variable */
   /* assume that data is in MMDDYYYY form, but check */
   /* Parse date string, get elements */
      cmo = SCAN(cdate,1,'/-.: ','M');
      cdy = SCAN(cdate,2,'/-.: ','M');
      cyr  = SCAN(cdate,3,'/-.: ','M');
      IF cyr EQ ' ' OR INDEX(cyr,'_') GT 0 OR cyr EQ '.' THEN
         y = .;
      ELSE
         y=INPUT(cyr,4.);

  /* Test year value - earliest date is 1990, cannot be in future */
      IF y ne . and (1990 LT y LT YEAR(TODAY())) THEN DO;
         /* assume year is good */
         IF cmo EQ ' ' OR INDEX(cmo,'_') GT 0 OR cmo EQ '.' THEN
```

```
            m = .;
         ELSE
            m=INPUT(cmo,2.);
         IF 1 LE m LE 12 THEN DO; /* month has to be between 1 and
12 */
           /* Month is good chk day */
            IF cdy EQ ' ' OR INDEX(cdy,'_') GT 0 OR cdy EQ '.' THEN
               d = .;
            ELSE
               d=INPUT(cdy,2.);
            IF d LE 1 OR d GT (DAY(INTNX('MONTH',MDY(m,1,y),0,'e')))
THEN DO;
               /* Day is bad for this month - impute middle of month
*/
               d = CEIL(DAY(INTNX('MONTH',MDY(m,1,y),0,'e'))/2);
               supr = '~{super Day}';
            END;
         END;
         ELSE DO;
            /* Month is bad - reset month and day */
            m=6;
            d=30;
            supr = '~{super Month}';
         END;
         /* Create the date value */
         date=MDY(m,d,y);
         dtstr = CATT(PUT(date,mmddyy10.),supr);
      END;
      ELSE DO;
         /* yr is bad -flag */
         dtstr = 'Unknown~{super Y}';
      END;
      RETURN(dtstr);
   ENDSUB;
RUN;
QUIT;
```

This section of the example will use the MAKEDATE() function created above inside of a custom format. It is important to note that the dates displayed are characters; therefore, the format is a character format (as indicated by the leading dollar sign [$]).The OTHER= keyword with no other format ranges defines a range of all possible values, and placing the function name inside the brackets (❷) applies the function.

Using the MAKEDATE() Function as a Custom Format

```
PROC FORMAT;
   VALUE $mkdt
     ❶ OTHER=[makedate()] ❷;
RUN;
```

```
DATA errorcnt;
INPUT recno @4 rptdt $10. errs;
DATALINES;
1  13/15/2013 374
2  09/08/2013 3
3  02/30/2013 32
4  __/15/2013 15
5  04/27/2013 195
6  05/__/2013 17
7  07/08/____ 20
8  06/04/013 5
9  08/32/2013 4
10 11//2013   6
11 01/23/2031 11
;;;;
RUN;

TITLE1 'Error Counts by Date';
PROC REPORT DATA=errorcnt NOWD SPLIT='\';
   COLUMN recno rptdt rptdt=dt errs;
   DEFINE recno / order 'Record #';
   DEFINE rptdt /  'Original\Date from\Error Log';
   DEFINE dt /  'Imputed\Date' f=$mkdt36.; ❶
   DEFINE errs / "Errors\Reported";
RUN;
```

The PROC REPORT step above uses the value that is stored in the data set and creates an alias (DT) for it. We apply the format we created to that aliased variable (❶) to show both the original and the on-the-fly imputation alias for it.

The Output

Record #	Original Date from Error Log	Imputed Date	Errors Reported
1	13/15/2013	06/30/2013 Month	374
2	09/08/2013	09/08/2013	3
3	02/30/2013	02/14/2013 Day	32
4	__/15/2013	06/30/2013 Month	15
5	04/27/2013	04/27/2013	195
6	05/__/2013	05/16/2013 Day	17
7	07/08/____	Unknown Y	20
8	06/04/013	Unknown Y	5

Record #	**Original Date from Error Log**	**Imputed Date**	**Errors Reported**
9	08/32/2013	08/16/2013^{Day}	4
10	11//2013	11/15/2013^{Day}	6
11	01/23/2031	Unknown^{Y}	11

It is important to realize that using this function as a format does *not* create imputed values in the data set. It is only a way to *display* the values from the data set.

2.8 The PUT() Function and Formats

What happens if you need to use the formatted value of a date in a character string you're assembling? If you use the variable that contains the date value, you'll get the actual SAS date value, regardless of any permanent formats assigned to the variable. The PUT() function is used to store the formatted value of a numeric value in a character variable. The syntax is:

PUT(*value,format***);**

value is a constant or a variable (either numeric or character), and *format* is the name of a SAS format. If you are formatting a character variable or constant, then the format you use must be a character format. Similarly, if you are formatting a numeric value, the format must be a numeric format. Example 2.11 will demonstrate how to use the PUT() function. The PUT() function ❶ creates the character variable *char_value*. The length of the variable in this case is defined by the width of the format, 10. If you've already defined the length of the character variable where you store the result, it has to be at least as long as the format width, or your output will be truncated.

Example 2.11: Using the PUT() Function to Create a Character Date String

```
DATA sample281;
numeric_value = 18947;
fmt_num_value = 18947;
char_value = PUT(numeric_value,MMDDYY10.); ❶
FORMAT fmt_num_value mmddyy10.;
LABEL numeric_value="SAS Date\Value"
      fmt_num_value="Formatted\SAS Date\Value"
      char_value="Character Date\in String";
RUN;

ODS RTF FILE="c:\book\2ndEd\examples\putfunction.rtf";

PROC PRINT DATA=sample281 NOOBS SPLIT='\';
RUN;
ODS RTF CLOSE;
```

SAS Date Value	Formatted SAS Date Value	Character Date in String
18947	11/16/2011	11/16/2011

So what's the difference? The formatted value looks the same as the character date, right? However, in this simple PROC PRINT, one difference is apparent. The numeric variable is right-justified (the default for the MMDDYY. format is right-justified), while the character variable is left-justified (default for character variables). If you look at the labels, you will see that the labels also reflect the default justification. The labels for the numeric variables are right-justified, and the one for the character variable is left-justified.

Why would you need to use a character representation of a SAS date? If you are printing a date inside of a text string (e.g., in a title or footnote, or in the text of a form letter), you will need to create a character variable with the date. Example 2.12 puts the RPTDATE. format created in Section 2.6.1 on each page of a report as a footnote.

Example 2.12: Using a Character Date String in a Footnote

```
PROC FORMAT;
PICTURE rptdate (DEFAULT=43)
 . - .Z = 'Missing'
LOW-HIGH = 'Generated on %B %d, %Y at %I:%0m %p'
(DATATYPE=DATETIME);
RUN;
```

In order to do this, we will have to create a macro variable to store the value so that it can be used in a FOOTNOTE statement. The method chosen here is via a DATA step and the CALL SYMPUTX statement. CALL SYMPUTX trims any trailing spaces as opposed to CALL SYMPUT. There are other ways to create the macro variable &RPTDTM as well (see Section 6.1.3).

```
DATA _NULL_;
CALL SYMPUTX('RPTDTM',PUT(DATETIME(),RPTDATE.));
RUN;
FOOTNOTE "&RPTDTM";
```

Chapter 3: Converting Dates and Times into SAS Date, Time, and Datetime Values

3.1 Avoiding the Two-Digit Year Trap.. 57

3.2 Using Informats.. 59

3.3 The INFORMAT Statement... 59

3.4 Listing and Discussion of Informats... 64

In Chapter 2, I discussed translating SAS dates to the way we express them. How can we do the reverse? After all, if you have a date, time, or date and time that you need to store or manipulate, it won't be represented as a SAS date, time, or datetime value (unless it comes from another SAS data set). The translation from common date and/or time terminology to SAS is almost as easy as going the other way, and it is done in one of two ways: The first, discussed in Section 1.4, uses date, time, and datetime literals. While this works for a small number of these values that are known at the time you write the program, how do you deal with multiple dates, or those that are known only at run time? By using *informats* to process them.

3.1 Avoiding the Two-Digit Year Trap

First, it is always good practice to use four-digit years in your date values. If this is not possible, SAS will handle dates with two-digit years without any problems, but this situation is where you, the user, should be concerned with the YEARCUTOFF= system option. Anytime that you translate a date or datetime from an everyday representation using two-digit years (for example, 05/16/89) to its SAS value, the YEARCUTOFF= system option will affect the resulting SAS value. The

YEARCUTOFF= system option will define the beginning of the 100-year period for those two-digit years. In SAS 9.4, the SAS default value for this option is 1926, defining a range of 1926 through 2025. Therefore, two-digit years in the range from 26 through 99 will be assigned to the years 1926–1999, while year values from 00 through 25 will be set to the years 2000–2025. This rule applies to everything that has to be translated into a SAS date or datetime value and where there are only two digits representing the year. This means that the YEARCUTOFF= option applies to external data being processed with the INPUT statement, any date or datetime literals, as well as any functions (such as INPUT()) where the input string only has two digits representing the year. The example below shows how date and datetime literals are affected by the YEARCUTOFF= system option. It displays the actual SAS date or datetime value represented by the literal along with its formatted value.

Example 3.1: Effects of the YEARCUTOFF= System Option on Date and Datetime Literals

OPTIONS YEARCUTOFF=1920

	SAS Date Literal	Value as Stored in SAS	Formatted Value
a1	'23MAR2005'd	16518	03/23/2005
a2	'23MAR1905'd	-20007	03/23/1905
a3	'23MAR05'd	16518	03/23/2005
a4	'19AUG1959:14:45:00'dt	-11610900	19AUG1959:14:45:00
a5	'19AUG2059:14:45:00'dt	3144149100	19AUG2059:14:45:00
a6	'19AUG59:14:45:00'dt	-11610900	19AUG1959:14:45:00

The variables a3 and a6 are date and datetime constants, respectively, with two-digit years as a part of the constant. While the literals with four-digit years are stored as their proper SAS dates, a3 is in the 21st century, but a6 is set in the 20th century. The 100-year range is 1920–2019. Therefore, '05' falls in the range 00–19 and is assigned to the year 2005, and '59' falls in the range 20–99 and is assigned to the year 1959. For the following example, let's use the same date and datetime literals, but change the value of the YEARCUTOFF= option.

OPTIONS YEARCUTOFF=1905

	SAS Date Literal	Value as Stored in SAS	Formatted Value
a1	'23MAR2005'd	16518	03/23/2005
a2	'23MAR1905'd	-20007	03/23/1905
a3	'23MAR05'd	-20007	03/23/1905
a4	'19AUG1959:14:45:00'dt	-11610900	19AUG1959:14:45:00

	SAS Date Literal	Value as Stored in SAS	Formatted Value
a5	'19AUG2059:14:45:00'dt	3144149100	19AUG2059:14:45:00
a6	'19AUG59:14:45:00'dt	-11610900	19AUG1959:14:45:00

Now, a3 has moved to the 20th century. The literals with four-digit years remain stored as their proper SAS dates and a6 remains in the 20th century, but we have moved the 100-year range to start in 1905. Therefore, the range is now 1905–2004, and '05' falls in the 05–99 range, so it is assigned to 1905.

The same thing will happen when you use an informat to translate a date or datetime string where there are only two digits representing the year.

3.2 Using Informats

Formats take values and display them in a specific fashion, while informats take a series of alphanumeric characters and translate them into a single value. Dates and times are the best example of applying informats, since SAS date values are not normally how the majority of the planet expresses dates. A date such as 11/24/05, or 24-05-2002 contains non-numeric characters, so they would have to be read as character values, which is quite a distance from numeric SAS date values. As with formats, you can create (and store for future use) your own informats if one is not available within SAS to fit your needs. There are fewer SAS-supplied informats than formats , so you should be careful to only use the informats listed in this chapter to translate your date and datetime strings, and not the formats from Chapter 2.

Just as you can use the FORMAT statement to apply a format to a variable, you can use the INFORMAT statement to apply an informat to a variable in a procedure or a DATA step. However, the most common use of informats is with the INPUT statement in a DATA step, as data are being read in, or with the INPUT() function to translate character data that are already in data sets.

You specify an informat by giving the informat name, followed by an optional width specification and a period (.). Informats are like formats in that each informat has a default width that SAS will use if none is specified.

3.3 The INFORMAT Statement

The INFORMAT statement is analogous to the FORMAT statement. You use the INFORMAT statement to associate an informat with a variable in a SAS data set. You can also remove an informat that has been permanently associated with the variable by leaving the informat name

blank. You might also use the INFORMAT statement to associate an informat with a variable for the duration of the procedure (which might be useful in certain procedures such as the FSEDIT procedure).

```
INFORMAT date1 mmddyy10.;
```

The statement above says that any time a character string is read into the variable date1, it will be translated into a SAS date value using the informat MMDDYY10 (detailed below). If the character string being processed is not in an MM***s***DD***s***YY(YY) (*s* stands for a separator), you will get a missing value stored in date1 and an error message in the SAS log.

Just as with formats, you can remove informats that have been permanently associated with a variable by using the variable name in the INFORMAT statement without an informat, as is shown below.

```
INFORMAT time3;
```

This statement will remove any informat that has been permanently associated with the variable time3.

3.3.1 Using Informats with the INPUT Statement

The basic syntax of an INPUT statement with an informat is as follows:

```
INPUT @1 date1 mmddyy10.;
```

First, you will usually specify a starting column. The default starting column is 1, but you can specify the starting column with the @ sign, followed by the column number. If you do not, the starting column will be set to the current location of the input pointer. You should also specify a width for the informat to indicate how many characters are to be read. The above INPUT statement will read the first ten characters in a line, starting at the first character in a data line, and SAS will expect it to look like mm*s*dd*s*(yy)yy, where mm is the month from 01 to 12, *s* represents a separator character, dd is a day from 01 to 31, and (yy)yy is a two- or four-digit year. The following example demonstrates that the separator character does not have to be the same on every line, and the field does not have to be exactly ten characters long.

Example 3.2: INPUT Statement

```
1. DATA informats_are_smart;
2. INPUT @1 date1 :MMDDYY10.;
3. unformatted_date = date1;
4. DATALINES;
5. 10/17/2014
6. 05-04-59
7. 3-1-1940
8. RUN;
```

```
9.  PROC PRINT;
10. FORMAT date worddatx.;
11. RUN;
```

The Result

Obs	date1	unformatted_date
1	17 October 2014	20013
2	4 May 1959	-242
3	1 March 1940	-7245

As you can see, the characters in each line of the DATALINES statement were converted to a SAS date value. The length of the informat must be long enough to read all of the characters in the date string. In the example, there are a maximum of 10 characters in the date string. Therefore, the width of the informat is specified as 10. The other important thing to note is the use of the colon (**:**, line 2, in **bold**) modifier preceding the informat name. It is best practice to use the colon modifier with *any* informat, not just those relating to dates and times, when you are dealing with varying length fields. The colon modifier ensures that it doesn't matter if the lengths of the character strings representing date in the data are less than the width specified for the informat.

If the characters read do not match that layout (for example, June 26, 2014, when the informat specified is MMDDYY.) or if the informat would yield an impossible value (for example, February 31), SAS will set the value of the variable date1 to missing and set the system variable _ERROR_ to 1. In general, you should know the layout of the characters before selecting an informat. Beginning with version 9 of SAS, if you do not know the layout of the dates ahead of time, the ANYDT family of informats (see Section 3.4.4) are designed to solve this problem.

3.3.2 Informats with the INPUT() Function

The INPUT() function is the parallel to the PUT() function, and it stores a numeric or character value as a numeric or character variable. The type of the result depends on the type of the informat that is used. A character informat (one that begins with a $) will return a character value. All informats used with dates, times, and datetimes are numeric. Therefore, the variable returned is numeric. The syntax is given below.

```
INPUT(character-value,informat-name);
```

If you want to define the informat that is to be applied during a SAS job (at run time), you will need to use the INPUTN(*character-value,informat-name*) function (or INPUTC(), if you want to produce a character variable) instead. *informat-name* represents a character variable or character constant that contains an informat name, while the INPUT() function needs an actual informat name. Make sure that you have defined the width of the informat so that it is long enough to

capture all the characters in the entire character variable. The following examples illustrate the use of the INPUT() and INPUTN() functions:

Example 3.3: INPUT() Function

```
DATA _NULL_ _;
datestr = "15-NOV-2013";
sasdate = INPUT(a,date11.);
PUT sasdate=;
RUN;
```

The INPUT() function translated the date in the character variable **datestr** into its equivalent SAS date value, 19677, and stored it in the numeric variable **sasdate**. The date11. informat accounts for the length of the character variable.

Example 3.4: INPUTN() Function

```
DATA _NULL_;
datestr = "15-NOV-2013";
inf = "DATE11.";
sasdate = INPUTN(a,inf);
PUT sasdate=;
RUN;
```

The INPUTN() function used the value of the character value **inf** (DATE11.) as the informat to use in translating the date in the character variable **datestr** into its equivalent SAS date value, 19677.

3.3.3 When the Informat Does Not Match the Data Being Read

Informats, like formats, are separated into classes according to the type of data that are being read. In most cases, if you use the wrong informat for the data type, informats will return an error (set the SAS automatic variable _ERROR_ to 1), and the value of the variable being read will be set to missing.

This behavior differs from date, time, and datetime formats in that if you use the wrong type of format to display a value (for example, if you use a date format to display a time value), no error will occur, and at worst, you will get a warning in the SAS log. However, incorrectly specifying a format will most likely cause the display to be incorrect. Example 3.5 shows what happens when you try to use an informat that does not match the character string that you are trying to process.

Example 3.5: Using the Wrong Informat

```
DATA bad_informat;
INPUT @1 date :datetime18.;
DATALINES;
11-06-1988
8-25-2004
4-24-2005
```

```
;;;;
RUN;
```

The Log

```
 1     DATA bad_informat;
 2     INPUT @1 date :datetime18.;
 3     DATALINES;

 NOTE: Invalid data for date in line 4 1-18.
 RULE:    ----+----1----+----2----+----3----+----4----+----5----+---
-6----+
 4        11-06-1988
 date=. _ERROR_=1 _N_=1
 NOTE: Invalid data for date in line 5 1-18.
 5        8-25-2004
 date=. _ERROR_=1 _N_=2
 NOTE: Invalid data for date in line 6 1-18.
 6        4-24-2005
 date=. _ERROR_=1 _N_=3
 NOTE: The data set WORK.BAD_INFORMAT has 3 observations and 1
variables.
 NOTE: DATA statement used (Total process time):
       real time           0.53 seconds
       cpu time            0.03 seconds
 7     ;;;;
 8     RUN;
```

The Resulting SAS Data set

Obs	date
1	.
2	.
3	.

As you can see in the above example, using the DATETIME. informat to process a series of character strings that do not represent datetimes produces a note in the log. It also sets the automatic variable _ERROR_ to 1 for each record where it encountered a mismatch between the informat specified and the data that it attempted to read. The end result is that the value of the date variable in your output data set is missing because SAS was not able to process the characters using the specified informat. Remember to always check your log and your data set after reading a text file.

3.4 Listing and Discussion of Informats

Each discussion of an informat in this section will provide an explanation of the informat, its width specification, and the text it is designed to process. Each subsection is accompanied by a table that gives examples of the text that is to be processed, along with the informat (and its width specification) and the resulting SAS date, time, or datetime value. SAS continues to develop formats and informats, so it is always a good idea to check the online documentation at support.sas.com.

3.4.1 Date Informats

DATE*w*.

DATE*w*. reads dates in the form *ddmonyy(yy)*, where *dd* represents the day of the month, *mon* is the three-letter month abbreviation, and *yy(yy)* is the two- or four-digit year. The default value of *w* is 7, but you should specify 9 if you are reading four-digit years. *dd*, *mon,* and *yy(yy)* can be separated by blanks or special characters. If you separate them, you must account for the blanks (or special characters) in the width specification. If you have blanks after the month and the day, then you need to have a width of 9 for two-digit years or 11 for four-digit years. If the leading zero for *dd* is missing, it has no effect on the value. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
20sep15	DATE7.	20351	Sunday, September 20, 2015
4 feb 2014	DATE11.	19758	Tuesday, February 4, 2014
07-may-1960	DATE11.	127	Saturday, May 7, 1960

DDMMYY*w.*

DDMMYY*w*. reads dates of the form *ddmmyy(yy)*, where *dd* represents the day of the month, *mm* represents the number of the month, and *yy(yy)* is the two- or four-digit year. The default value of *w* is 6, but you should specify 8 if you are reading four-digit years. *dd, mm, and yy(yy)* can be separated by blanks or special characters. If you separate them, you must account for the separating characters in the width specification. If you have blanks after the month and the day, then you need to have a width of 8 for two-digit years or 10 for four-digit years. SAS will do its best to decipher the string if no separators are used, but some dates cannot be processed—for example, 2112008. Without place-holding zeros or separators, there is no way to know whether the date is 21 January 2008 or 2 November 2008. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
140390	DDMMYY6.	11030	Wednesday, March 14, 1990
06/09/05	DDMMYY8.	16685	Tuesday, September 6, 2005
22-04-2003	DDMMYY10.	15817	Tuesday, April 22, 2003

JULIAN*w*.

JULIAN*w*. translates a Julian date in the form *yy(yy)ddd*, with the two- or four-digit year preceding the zero-filled day of the year. It is right-justified. *w* can be from 5 to 32, and the default is 5. If you specify 5, the year portion of the Julian date is two digits long. If you specify 7 or more, the year portion is four digits long. Zeros must fill the space between the year and day values: For example, the fifth day of the year must be given as "005." Any date preceding the year 1582 on the Gregorian calendar cannot be read as a Julian value. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
09284	JULIAN5.	18181	Sunday, October 11, 2009
2014005	JULIAN7.	19728	Sunday, January 5, 2014
2012168	JULIAN7.	19160	Saturday, June 16, 2012

MMDDYY*w*.

MMDDYY*w*. reads dates of the form *mmddyy(yy)*, where *mm* represents the number of the month, *dd* represents the day of the month, and *yy(yy)* is the two- or four-digit year. The default value of *w* is 6, but you should specify 8 if you are reading four-digit years. *dd*, *mm*, and *yy(yy)* can be separated by blanks or special characters. If you separate them, you must account for the blanks in the width specification. If you have blanks after the month and the day, then you need to have a width of 8 for two-digit years or 10 for four-digit years. SAS will do its best to decipher the string if no separators are used, but some dates cannot be processed—for example, 1272003. Without place-holding zeros or separators, there is no way to know whether the date is January 27, 2003, or December 7, 2003. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
041798	MMDDYY6.	13986	Friday, April 17, 1998
1/15/2013	MMDDYY10.	19373	Tuesday, January 15, 2013

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
08282015	MMDDYY10.	20328	Friday, August 28, 2015

MONYY*w*.

MONYY*w*. reads dates of the form *monyy(yy)*, where *mon* is the three-letter month abbreviation, and *yy(yy)* is the two- or four-digit year. Using this informat will set the SAS date value that corresponds to the first day of the month. The default value of *w* is 5, but you should specify 7 if you are reading four-digit years. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
JAN15	MONYY5.	20089	Thursday, January 1, 2015
dec1920	MONYY7.	-14275	Wednesday, December 1, 1920
aug2020	MONYY7.	22128	Saturday, August 1, 2020

PDJULG4.

PDJULG4. reads a packed Julian date in hexadecimal format for IBM computers. The width specification is always 4, because the Julian date is parsed as follows: the four-digit Gregorian year is written in the first two bytes, and the three-digit integer that represents the day of the year is in the next one-and-a-half bytes. The last half-byte contains all binary 1s, which indicates the value is positive. There is no example given for this informat because packed-decimal Julian dates yield nonprintable characters.

PDJULI*w*.

PDJULI*w*. also reads a packed Julian date in hexadecimal format for IBM computers. It differs from the PDJULG. informat in that it expects the two digits of the century in the first byte, followed by two digits of the year in the second byte. The next one-and-a-half-bytes store the three-digit integer that corresponds to the day of the year, while the last half-byte is filled with hexadecimal 1s, representing a positive number. The century and year are calculated by subtracting 1900 from the four-digit Gregorian year. Once again, there is no example, since packed-decimal Julian dates yield nonprintable characters.

YYMMDD*w*.

YYMMDD*w*. is used to read dates of the form *yy(yy)mmdd*, where *yy(yy)* is the two- or four-digit year, *mm* represents the number of the month, and *dd* represents the day of the month. The default value of *w* is 6, but you should specify 8 if you are reading four-digit years. *yy(yy)*, *mm*, and *dd* can be separated by blanks or special characters. If you separate them, you must account for the

separating characters in the width specification. If you have blanks after the month and the day, then you need to have a width of 8 for two-digit years or 10 for four-digit years. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
441205	YYMMDD6.	-5505	Tuesday, December 5, 1944
20150517	YYMMDD8.	20225	Sunday, May 17, 2015
2014-07-06	YYMMDD10.	19910	Sunday, July 6, 2014

YYMMN*w*.

YYMMN*w*. reads dates of the form *yy(yy)mm*, where *yy(yy)* is the two- or four-digit year, and *mm* represents the number of the month. The day is automatically set to 1. The default value of *w* is 4, but you should specify 6 if you are reading four-digit years. The *N* in the informat name is necessary. Your data must not have any separating characters between the month and the year. This informat will produce a date value that is equal to the first day of the month given. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line. Note that line 2 of this table demonstrates what happens if you try to read a date with a two-digit year using the width of 6. In this case, SAS is translating the four digits as the year, so the month value is considered to be missing.

	Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
1	9905	YYMMN4.	14365	Saturday, May 1, 1999
2	***9905***	***YYMMN6.***	**.**	
3	201403	YYMMN6.	19783	Saturday, March 1, 2014
4	201610	YYMMN6.	20728	Saturday, October 1, 2016

YYQ*w*.

YYQ*w*. reads dates of the form *yy(yy)Qq*, where *yy(yy)* is the two- or four-digit year followed by the letter Q and *q* is a number from 1 to 4, indicating the quarter of the year. The date value produced by this informat will correspond to the first day of the given quarter. Use 6 for *w* if you are reading four-digit years, or 4 if you are reading two-digit years. The default *w* is 6. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
14Q1	YYQ4.	19724	Wednesday, January 1, 2014
2010Q3	YYQ6.	18444	Thursday, July 1, 2010
2015Q2	YYQ6.	20179	Wednesday, April 1, 2015
2013Q4	YYQ6.	19632	Tuesday, October 1, 2013

YYMMDD*w*.

YYMMDD*w*. is used to read dates of the form *yy(yy)mmdd*, where *yy(yy)* is the two- or four-digit year, *mm* represents the number of the month, and *dd* represents the day of the month. The default value of *w* is 6, but you should specify 8 if you are reading four-digit years. *yy(yy)*, *mm*, and *dd* can be separated by blanks or special characters. If you separate them, you must account for the separating characters in the width specification. If you have blanks after the month and the day, then you need to have a width of 8 for two-digit years or 10 for four-digit years. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
441205	YYMMDD6.	-5505	Tuesday, December 5, 1944
20150517	YYMMDD8.	20225	Sunday, May 17, 2015
2014-07-06	YYMMDD10.	19910	Sunday, July 6, 2014

YYMMN*w*.

YYMMN*w*. reads dates of the form *yy(yy)mm*, where *yy(yy)* is the two- or four-digit year, and *mm* represents the number of the month. The day is automatically set to 1. The default value of *w* is 4, but you should specify 6 if you are reading four-digit years. The *N* in the informat name is necessary. Your data must not have any separating characters between the month and the year. This informat will produce a date value that is equal to the first day of the month given. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line. Note that line 2 of this table demonstrates what happens if you try to read a date with a two-digit year using the width of 6. In this case, SAS is translating the four digits as the year, so the month value is considered to be missing.

	Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
1	9905	YYMMN4.	14365	Saturday, May 1, 1999
2	***9905***	***YYMMN6.***	**.**	
3	201403	YYMMN6.	19783	Saturday, March 1, 2014

	Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
4	201610	YYMMN6.	20728	Saturday, October 1, 2016

YYQ*w*.

YYQ*w*. reads dates of the form *yy(yy)Qq*, where *yy(yy)* is the two- or four-digit year followed by the letter *Q* and *q* is a number from 1 to 4, indicating the quarter of the year. The date value produced by this informat will correspond to the first day of the given quarter. Use 6 for *w* if you are reading four-digit years, or 4 if you are reading two-digit years. The default *w* is 6. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
14Q1	YYQ4.	19724	Wednesday, January 1, 2014
2010Q3	YYQ6.	18444	Thursday, July 1, 2010
2015Q2	YYQ6.	20179	Wednesday, April 1, 2015
2013Q4	YYQ6.	19632	Tuesday, October 1, 2013

The WEEK Informats

The WEEK informats read dates in the ISO-standard WEEK format. There are three algorithms used to calculate the WEEK value from a given date. The U algorithm calculates weeks based on Sunday being the first day of the week without any other restriction. The V algorithm defines the first week of the year as containing both January 4 and the first Thursday of the year. Therefore, if the first Monday of the year falls on January 2, 3, or 4, the preceding days of the calendar year are considered to be a part of week 53 of the previous calendar year. It is also possible for calendar days at the end of a year to be considered as being in the first week of the next calendar year. Finally, the W algorithm calculates weeks based on Monday being the first day of the week without any other restriction.

The WEEK algorithms calculate dates differently, so it is critical that you use the correct algorithm for the week value that you are converting. This means that you need to know which algorithm was used to create the week value before you try to convert it. If you use a different week algorithm from the one that was used to create the week value, you will get the wrong actual date. Example 3.6 demonstrates this.

Example 3.6: The Difference between the Various WEEK Algorithms (U, V, W)

	Week value	Date Formatted with WEEKDATE U Algorithm	Date Formatted with WEEKDATE V Algorithm	Date Formatted with WEEKDATE W Algorithm
1	W12	Sunday, March 23, 2014	Monday, March 17, 2014	Monday, March 24, 2014
2	13W45	Sunday, November 10, 2013	Monday, November 4, 2013	Monday, November 11, 2013
3	15W5303	Tuesday, January 5, 2016	Wednesday, December 30, 2015	Wednesday, January 6, 2016
4	2014W2104	Wednesday, May 28, 2014	Thursday, May 22, 2014	Thursday, May 29, 2014
5	2015-W01-02	Monday, January 5, 2015	Tuesday, December 30, 2014	Tuesday, January 6, 2015

Example 3.6 demonstrates the differences that exist not only in the resulting date between the algorithms, but also in the day of the week, which depends on whether the algorithm uses Sunday or Monday as the first day of the week. Rows 3 and 5 highlight the differences in the way the first and last week of the calendar year are handled by the different algorithms. As you can see, if you use the wrong algorithm, not only can you end up with an unintended date, but the resulting date might not even be in the same year.

WEEKU*w*.

WEEKU*w*. is new as of SAS version 9.3. It is used to translate week values into SAS dates using the U algorithm. The U algorithm calculates weeks based on Sunday being the first day of the week, and the week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. Week values are dates in the form *yyyy-Wnn-dd*, where *yyyy* is the optional year (can be two or four digits). *W* is the letter "W" for "Week"; *nn* is the week number ranging from 0 to 53; and *dd* is the optional day of the week. The dashes are optional separators, but they are the only separators that can be used in ISO 8601 week values and are used between the year, the week number, and the day (if present). The width specification, *w*, can range from 3 to 200 and tells SAS what to expect from the character string being processed according to the following table.

Width (*w*)	Pattern Expected	Example	Comment
3–4	*W*nn	W08	*W* is the letter "W," and *nn* is the week number. If a year is not specified, the year is considered to be the current year. If a day is not given, the day is considered to be the first day of the week.
5–6	yy*W*nn	14W08	*yy* is a two-digit year, *W* is the letter "W," and *nn* is the week number. Without a day, the day is considered to be the first day of the week.
7–8	yy*W*nndd	14W0803	*yy* is a two-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week.

Width (*w*)	Pattern Expected	Example	Comment
9–10	yyyy*W*nndd	2014W0803	*yyyy* is a four-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week.
11–200	yyyy-*W*nn-dd	2014-W08-03	*yyyy* is a four-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week, each separated by dashes.

Specifying any value greater than 11 will have no effect on the date returned, although the implied cursor placement can cause an INPUT statement (not function) to yield unexpected results.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
W12	WEEKU3.	19805	Sunday, March 23, 2014
14W40	WEEKU5.	20001	Sunday, October 5, 2014
15W2801	WEEKU7.	20281	Sunday, July 12, 2015
2014W3304	WEEKU9.	19955	Wednesday, August 20, 2014
2015-W06-02	WEEKU11.	20128	Monday, February 9, 2015

WEEKV*w*.

WEEKV*w*. is available as of SAS version 9.3. It is used to translate week values into SAS dates using the V algorithm. The V algorithm calculates weeks based on Monday being the first day of the week, and the week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. This algorithm defines the first week of the year as containing both January 4 and the first Thursday of the year. Therefore, if the first Monday of the year falls on January 2, 3, or 4, the preceding days of the calendar year are considered to be a part of week 53 of the previous calendar year. Week values are dates in the form *yyyy-Wnn-dd*, where *yyyy* is the optional year (can be two or four digits). *W* is the letter "W" for "Week"; *nn* is the week number ranging from 0 to 53; and *dd* is the optional day of the week. The dashes are optional separators, but they are the only separators that can be used in ISO 8601 week values and are used between the year, the week number, and the day (if present). The width specification, *w*, can range from 3 to 200 and tells SAS what to expect from the character string being processed according to the following table:

Width (*w*)	Pattern Expected	Example	Comment
3–4	*W*nn	W08	*W* is the letter "W," and *nn* is the week number. If a year is not specified, the year is considered to be the current year. If a day is not given, the day is considered to be the first day of the week.

Width (*w*)	Pattern Expected	Example	Comment
5–6	yy*W*nn	14W08	*yy* is a two-digit year, *W* is the letter "W," and *nn* is the week number. Without a day, the day is considered to be the first day of the week.
7–8	yy*W*nndd	14W0803	*yy* is a two-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week.
9–10	yyyy*W*nndd	2014W0803	*yyyy* is a four-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week.
11–200	yyyy-*W*nn-dd	2014-W08-03	*yyyy* is a four-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week, each separated by dashes

Specifying any value greater than 11 will have no effect on the date returned, although the implied cursor placement can cause an INPUT statement (not function) to yield unexpected results.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
W12	WEEKV3.	19799	Monday, March 17, 2014
14W40	WEEKV5.	19995	Monday, September 29, 2014
15W2801	WEEKV7.	20275	Monday, July 6, 2015
2014W3304	WEEKV9.	19949	Thursday, August 14, 2014
2015-W06-02	WEEKV11.	20122	Tuesday, February 3, 2015

WEEKW*w*.

WEEKW*w*. is available as of SAS version 9.3. It is used to translate week values into SAS dates using the W algorithm. The W algorithm calculates weeks based on Monday being the first day of the week without any other restriction. The week number is displayed as a two-digit number from 0 to 53, with a leading zero if necessary. Week values are dates in the form *yyyy-Wnn-dd*, where *yyyy* is the optional year (can be two or four digits). *W* is the letter "W" for "Week"; *nn* is the week number ranging from 0 to 53; and *dd* is the optional day of the week. The dashes are optional separators, but they are the only separators that can be used in ISO 8601 week values and are used between the year, the week number, and the day (if present). The width specification, *w*, can range from 3 to 200 and tells SAS what to expect from the character string being processed according to the following table:

Width (*w*)	Pattern Expected	Example	Comment
3–4	*W*nn	W08	*W* is the letter "W," and *nn* is the week number. If a year is not specified, the year is considered to be the current year. If a day is not given, the day is considered to be the first day of the week.
5–6	yy*W*nn	14W08	*yy* is a two-digit year, *W* is the letter "W," and *nn* is the week number. Without a day, the day is considered to be the first day of the week.
7–8	yy*W*nndd	14W0803	*yy* is a two-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week.
9–10	yyyy*W*nndd	2014W0803	*yyyy* is a four-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week.
11–200	yyyy-*W*nn-dd	2014-W08-03	*yyyy* is a four-digit year, *W* is the letter "W," *nn* is the week number, and *dd* is the day of the week, each separated by dashes.

Specifying any value greater than 11 will have no effect on the date returned, although the implied cursor placement can cause an INPUT statement (not function) to yield unexpected results.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WEEKDATE.
W12	WEEKW3.	19806	Monday, March 24, 2014
14W40	WEEKW5.	20002	Monday, October 6, 2014
15W2801	WEEKW7.	20282	Monday, July 13, 2015
2014W3304	WEEKW9.	19956	Thursday, August 21, 2014
2015-W06-02	WEEKW11.	20129	Tuesday, February 10, 2015

3.4.2 Time Informats

HHMMSS*w*.

HHMMSS*w*. is available as of SAS version 9.3. It will read time values in the form *hh:mm:ss* or *hhmmss*, where *hh* is hours, *mm* is minutes, and *ss* is seconds, with an optional separator. This informat will ignore fractional seconds. The default width for this format is 8, but *w* can range from 1 to 20. When there are six digits in the string being read, the first two digits will be translated into hours, the second two into minutes, and the last two into seconds.

Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME8.
143500	HHMMSS6.	52500	14:35:00
971406	HHMMSS6.	350046	97:14:06
000339	HHMMSS6.	219	0:03:39
081525	HHMMSS6.	29725	8:15:25

The parsing and interpretation performed by this informat changes if there are fewer than six digits in the string being read and is based on the following rules: If there is an odd number of digits in the string being read, SAS will add a zero to the beginning of the string, and the first two digits will then be translated as hours. SAS will then add zeros to the end of the string until it has six digits and can be translated as hours, minutes, seconds.

Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME8.	What Happened?
1	HHMMSS6.	3600	1:00:00	Because there is an odd number of digits in the string, SAS added a zero to the front of the string, making it "01," and then padded it with zeros out to six digits, making the string that is being read "010000."
11	HHMMSS6.	39600	11:00:00	Here, there are an even number of digits, so SAS does not add a leading zero and reads the first two digits as hours. It pads the remainder of the string with zeros out to six digits, making the string "110000."
112	HHMMSS6.	4320	1:12:00	With the odd number of digits, SAS adds a zero to the beginning of the string, making it "0112," and then pads the string with zeros, so the string being read is "011200."
1127	HHMMSS6.	41220	11:27:00	At four digits, SAS will only pad the string with zeros, making the string being read "112700."

Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME8.	What Happened?
11274	HHMMSS6.	4394	***1:13:14***	SAS adds a leading zero because of the odd number of digits in the string. The string that is being read is now "011274." This is translated into one hour, twelve minutes, and seventy-four seconds, which is 1:13:14.
112745	HHMMSS6.	41265	11:27:45	String being read is six digits long. Therefore, no zeros are added at the front or the back of the string.

When there are more than six digits in the string that is being read, SAS will parse the string from *right to left*. The assumption is that since seconds and minutes normally cycle at 60, they will only ever require two digits. It will translate the last four digits of the string as minutes and seconds. All digits to the left of the final four digits will be translated into hours.

Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME16.	What Happened?
172154	HHMMSS10.	62514	17:21:54	Parsed as hours, minutes, seconds. Result is as expected.
1721543	HHMMSS10.	620143	172:15:43	Last four digits are "1543," which is translated as 15 minutes, 43 seconds. Preceding digits ("172") translated as hours.
17215430	HHMMSS10.	6198870	1721:54:30	Last four digits are "5430," which is translated as 54 minutes, 30 seconds. Preceding digits ("1721") translated as hours.
172154305	HHMMSS10.	61976585	17215:43:05	
1721543058	HHMMSS10.	619756258	172154:30:58	

MSEC8.

MSEC8. reads IBM mainframe time values accurate to the nearest millisecond. The width is 8 because the OS TIME macro and STCK system instructions store their time values in 8 bytes.

PDTIME4.

PDTIME4. converts packed-decimal time values contained in SMF and RMF records produced by IBM mainframe systems to SAS time values. The width is shown as 4 because SMF and RMF records are 4 bytes long. While the informat RMFSTAMP8. also reads packed decimal RMF records, RMFSTAMP8. reads both the 4 bytes of time and 4 bytes of date information contained in the RMF record and creates a SAS datetime value from it. PDTIME4. only reads the 4 bytes of time information from an RMF record and creates a SAS time value.

RMFDUR4.

RMFDUR4. converts IBM mainframe RMF duration records into SAS time values. The width is shown as 4 because RMF records are 4-byte-long packed hexadecimal records.

STIMER*w*.

STIMER*w*. reads times produced by the STIMER System option in the SAS log. This informat has no default width. It reads times and interprets them based on colons and decimal points. If there is one colon, the first two digits are minutes and the last two are seconds. If there are two colons, the digits preceding the first colon are hours, the next set of two digits is minutes, and the last two are seconds. If there is a decimal point, the value following the decimal point is translated as a decimal fraction of seconds. It can read time values in the following formats, where *hh* corresponds to hours, *mm* corresponds to minutes, *ss* corresponds to seconds, and *ff* corresponds to decimal fractions of seconds.

```
ss
ss.ff
mm:ss
mm:ss.ff
hh:mm:ss
hh:mm:ss.ff
```

Characters Read	Informat	Resulting SAS Time Value (seconds)	Formatted Time Using TIME11.1.	Comments
33	STIMER2.	33.00	0:00:33.00	When there is no colon in the input string, it is translated as *ss*.
51.60	STIMER5.	51.60	0:00:51.60	The decimal point causes the translation to be *ss.ff*.

Characters Read	Informat	Resulting SAS Time Value (seconds)	Formatted Time Using TIME11.1.	Comments
14:05	STIMER5.	845.00	0:14:05.00	One colon is interpreted as *mm:ss.*
3:11.03	STIMER7.	191.03	0:03:11.03	One colon, and a decimal point is translated as *mm:ss.ff.*
1:19:21	STIMER7.	4761.00	1:19:21.00	Two colons are translated as *hh:mm:ss.*
1:46:17.74	STIMER11.	6377.74	1:46:17.74	Two colons and a decimal point, and the translation is *hh:mm:ss.ff.*

TIME*w*.

TIME*w*. will read times in the form *hh:mm:ss.ff*, where *hh* indicates the hours, *mm* is minutes, and *ss* is the number of seconds. They must be separated by a special character, such as a colon (:), period (.), or hyphen (-). *ff* indicates decimal fractions of seconds and must be separated from the seconds by a decimal point (.). Both seconds and their decimal fractions are assumed to be zero if they are not present. This informat can read a.m. and p.m. time values. If *hh* is greater than 24, and/or *mm* and *ss* are greater than 60, the time value read will give the correct number of seconds, even if it is greater than 86399.99 (the number of seconds in a day). This informat will parse the time string being read as hours, minutes, seconds from left to right. Therefore, if you are attempting to read only minutes and seconds, you must have leading zeros for the hours, and values for both minutes and seconds. Otherwise, your value will be translated as hours:minutes, not minutes:seconds. *w* ranges from 5 to 32 with a default of 8.

	Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME11.1.	Comments
1	124:46	TIME6.	449160.0	124:46:00.0	124 hours, 46 minutes
2	14:11:03.3	TIME10.	51063.3	14:11:03.3	
3	08-15	TIME5.	29700.0	8:15:00.0	
4	00:10	TIME5.	600.0	0:10:00.0	parsed as hours:minutes
5	00:10:42	TIME8.	642.0	0:10:42.0	parsed as hours, minutes, seconds
6	10.42	TIME10.	38520.0	10:42:00.0	parsed as hours:minutes despite the period separator
7	00:10:42.5	TIME10.	642.5	0:10:42.5	hours:minutes:seconds.fractional seconds

TODSTAMP8.

TODSTAMP8. converts an eight-byte time-of-day stamp produced by the OS TIME macro or the STCK instruction on IBM mainframes into a SAS time value. The width is 8 because these calls return eight-byte time-of-day values. Use this informat when you are reading IBM mainframe time-of-day values on other operating systems.

TU4.

TU4. converts IBM mainframe timer units (38,400 timer units per second) to SAS time values. It is used when reading IBM mainframe timer unit values under other operating systems. The width is 4 because the OS TIME macro returns a four-byte word.

3.4.3 Datetime Informats

B8601CI*w.d*

B8601CI*w.d* is available as of SAS version 9.3. It reads IBM time values with a century marker of the form *cyymmddhhmmss<fff>*, where *c* represents the century digit. The century digit is calculated by subtracting 1900 from the current year, dividing by 100, and dropping the remainder. *yy* is the two-digit year from 00 to 99, the *mm* represents the number of the month, and *dd* represents the day of the month. The time is represented by *hhmmss<fff>*, where *hh* indicates the hours, *mm* is minutes, *ss* is the number of seconds, and *fff* indicates thousandths of seconds. *w* ranges from 10 to 26, with a default value of 16, while *d* ranges from 0 to 6 for the fractional part of seconds.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME19.3.
21504231905	B8601CI16.	4901108700.000	23APR15:19:05:00.00
1560928053505	B8601CI16.	3052964105.000	28SEP56:05:35:05.00
2140630102416454	B8601CI19.3	4875416656.454	30JUN14:10:24:16.45
2131216094500	B8601CI16.	4858479900.000	16DEC13:09:45:00.00

B8601DJ*w.d*

B8601DJ*w.d* is available as of SAS version 9.3. It reads datetimes in standard Java date and time notation. *yyyymmddhhmmss<ffffff>*, where *yyyy* is the four-digit year, *mm* represents the number of the month, *dd* represents the day of the month, and time is represented by *hhmmss<ffffff>*, where *hh* indicates the hours, *mm* is minutes, *ss* is the number of seconds, and *ffffff* indicates millionths of seconds. *w* ranges from 10 to 26, with a default value of 16, while *d* ranges from 0 to 6 for the fractional part of seconds. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME19.5.
201607181108	B8601DJ16.	1784459280.0	18JUL16:11:08:00.00000
20141123054509	B8601DJ16.	1732340709.0	23NOV14:05:45:09.00000
201303070814433064	B8601DJ21.4	1678263283.3064	07MAR13:08:14:43.30640
201406241630254	B8601DJ16.1	1719246625.4	24JUN14:16:30:25.40000

DATETIME*w.d*

DATETIME*w.d* reads SAS datetime values. The datetime value must be in the form *ddmonyy(yy)*, followed by a blank or a special character, and then the time in the format *hh:mm:ss.ff*. *dd* represents the day of the month, *mon* is the three-letter month abbreviation, and *yy(yy)* is the two- or four-digit year. *hh* indicates the number of hours, *mm* is the number of minutes, and *ss* is the number of seconds. *ff* indicates fractional parts of seconds. Both seconds and fractional seconds are assumed to be zero if they are not present. *w* can be from 13 to 40, with a default of 18, while *d* can range from 0 to 6.

If you use a two-digit year, SAS will apply the YEARCUTOFF= system option in translating the year. This informat can also read a.m. and p.m. time values.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME22.1.
22APR2014 5:23 PM	DATETIME18.	1713806580.0	22APR2014:17:23:00.0
22APR2014-17:23	DATETIME16.	1713806580.0	22APR2014:17:23:00.0
22APR2014:05:23:15 PM	DATETIME22.	1713806595.0	22APR2014:17:23:15.0
22APR2014/17:23:15.6	DATETIME21.1	1713806595.6	22APR2014:17:23:15.6

MDYAMPM*w.d*

MDYAMPM*w.d* reads datetime values in the form of *mm-dd-yy(yy) hh:mm:ss.ff AM|PM*, where *mm* represents the number of the month, *dd* represents the day of the month, and *yy(yy)* is the two- or four-digit year, followed by the time where *hh* represents hours, *mm* represents minutes, *ss* represents optional seconds, *ff* represents optional fractional seconds, and AM|PM indicates a.m. or

p.m. The separators are *not* optional and can be a hyphen (-), or period (.), or slash (/), or colon (:). The default value of *w* is 19 and ranges from 8 to 40, while *d* ranges from 0 to 39 for the fractional part of seconds. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line. Line 4 shows that this format can translate twenty-four-hour clock values without the AM|PM indicator.

	Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME19.2.
1	05-08-2015 9:33 AM	MDYAMPM18.	1746696780.00	08MAY15:09:33:00.00
2	10-26-2014 5:00 PM	MDYAMPM18.	1729962000.00	26OCT14:17:00:00.00
3	04-25-2013 1:00:57.4 PM	MDYAMPM22.3	1682514057.40	25APR13:13:00:57.40
4	08-09-2017 14:15	MDYAMPM17.	1817907300.00	09AUG17:14:15:00.00

RMFSTAMP8.

RMFSTAMP8. converts IBM mainframe RMF date and time records into SAS datetime values. The width is shown as 8 because RMF records are packed-decimal records with 4 bytes of time information followed by 4 bytes of date information. While PDTIME4. also reads RMF records, it only reads the time portion of the RMF record and produces a SAS time value. RMFSTAMP8. reads both the date and time portions of the RMF record and produces a SAS datetime value.

SHRSTAMP8.

SHRSTAMP8. converts IBM mainframe SHR date and time records into SAS datetime values. The width is shown as 8 because SHR records are packed-decimal records with 4 bytes of date information followed by 4 bytes of time information.

SMFSTAMP8.

SMFSTAMP8. converts IBM mainframe SMF records into SAS datetime values. The width is shown as 8 because SMF records are packed-decimal records with 4 bytes of time information followed by 4 bytes of date information. This enables you to read SMF records without regard to operating system.

YMDDTTM*w.d*

YMDDTTMw.d reads datetime values in the form *<yy>yy-mm-dd hh:mm:ss.ff*, where special characters such as a hyphen (-), period (.), slash (/), or colon (:) are used to separate the year, month, day, hour, minute, and seconds. Separators are required for each component. The year can be either two or four digits. *yy(yy)* is the two- or four-digit year, *mm* represents the number of the month, and *dd* represents the day of the month. The time follows, where *hh* represents hours, *mm* represents minutes, *ss* represents optional seconds, and *ff* represents optional fractional seconds, which must be separated from the seconds by a decimal point (.). This informat will translate a.m. and p.m. time values. The default value of *w* is 19 and can range from 13 to 40, while *d* ranges from 0 to 39 for the fractional part of seconds.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME22.1.
2014-07-18-11-08-05	YMDDTTM19.	1721300885.0	18JUL2014:11:08:05.0
2014/01/11:05:19:45	YMDDTTM19.	1705036785.0	11JAN2014:05:19:45.0
1983.02.18.05:59 PM	YMDDTTM20.	730058340.0	18FEB1983:17:59:00.0
2015-08-23-14:24:16.5	YMDDTTM22.1	1755959056.5	23AUG2015:14:24:16.5

3.4.4 The "ANYDATE" Series of Informats

SAS 9 addressed an issue with the processing of dates and times that has always affected SAS users. Although informats handle the translation of a string of characters into SAS date and time values easily, in order to use them you had to know what the string of characters looked like before you processed them. Given the many ways that dates and times can be represented, it was not uncommon for several records to have incorrect values after processing because of an error in the underlying text strings being translated. For example, if you expect the dates to be in the form "ddMOMyyyy," but halfway through the file the strings were entered as "mmddyyyy," at least half of your resulting data set will have missing date values. The DATE. informat cannot read strings formatted to be read with the MMDDYY. informat, and vice versa. There are now three informats that will intelligently and, for the most part, successfully enable you to avoid this problem.

The ANYDTDTE., ANYDTDTM., and ANYDTTME. informats will translate dates, datetime values, *and* time values, respectively, into their corresponding SAS values. This translation will be performed without having to know the representation of these date, datetime, and time values in advance. Some limits exist as to the types of representations these informats will be able to translate, and in this era of big data you should also be aware that using these informats will require more CPU time than if you are able to use one of the regular informats to process your data.

The potential for confusion exists with DDMMYY, MMDDYY, and YYMMDD values, especially in the presence of two-digit year values. The SAS option DATESTYLE indicates how such confusions will be resolved. The possible values for the DATESTYLE= system option are shown in the following table.

Value	Explanations
MDY	Sets the default order as month, day, year. "10-03-12" would be translated as October 3, 2012.
YMD	Sets the default order as year, month, day. "10-03-12" would be translated as March 12, 2010.
DMY	Sets the default order as day, month, year. "10-03-12" would be translated as March 10, 2012.
LOCALE (default)	Sets the default value according to the LOCALE= system option. When the default value for the LOCALE= system option is "English_US," this sets DATESTYLE to MDY. Therefore, by default, "10-03-12" would be translated as October 3, 2012.

Note: YDM, MYD, and DYM have been removed from SAS 9.3 and later versions. If you continue to use them in legacy code, no error will appear in the log, but any input strings that cannot be translated without knowing the DATESTYLE (for example, "10-03-12") will be translated as missing.

What happens when the input string cannot be translated using the DATESTYLE option in effect? The ANYDT series of informats will test the input string to see whether one of the other DATESTYLE options will work. If only one DATESTYLE produces a valid SAS date or datetime value, then SAS will process the input string using that DATESTYLE. However, it is entirely possible that the input string can be translated using more than one of the remaining DATESTYLE options. For example, the input string "270328" cannot be translated using MDY, but both YMD and DMY will produce a valid SAS date. SAS has an internal priority of which DATESTYLE takes precedence when the option in effect will not work, and two or more DATESTYLES will. It is based on the DATESTYLE option in effect. The following table details how the DATESTYLE is chosen if the DATESTYLE option in effect does not yield a valid SAS date value.

DATESYTLE Option in Effect	Input String	DATESTYLEs Possible	DATESTYLE Used	Result
MDY	170514	YMD, DMY	YMD	May 14, 2017
YMD	110845	DMY, MDY	DMY	August 11, 1945
DMY	*No ambiguity is possible. DMY will always take precedence.*			

In short, when the DATESTYLE is MDY, then any ambiguity will be resolved using YMD, and when the DATESTYLE is YMD, any ambiguity is resolved using DMY.

ANYDTDTEw.

ANYDTDTEw. will translate data that can be read with the following informats: DATE, DATETIME, DDMMYY, JULIAN, MDYAMPM, MMDDYY, MMxYY, MONYY, TIME, YMDDTTM, YYMMDD, YYMMn, or YYQ into SAS date values. This informat can also work with the following standard date formats: "February 25, 2012," "08/2014," or "2015-05." w can range from 5 to 32, and the default width is 9. This informat extracts date values from a datetime string. However, if only a time value is given, the date is assumed to be January 1, 1960. The following table uses OPTIONS DATESTYLE=MDY as the default interpretation for two-digit year values.

Characters Read	Informat	Resulting SAS Date Value	Formatted Date Using WORDDATE.	Comments
05172014	ANYDTDTE8.	19860	May 17, 2014	
20140517	ANYDTDTE8.	19860	May 17, 2014	

Characters Read	**Informat**	**Resulting SAS Date Value**	**Formatted Date Using WORDDATE.**	**Comments**
2014Q2	ANYDTDTE6.	19814	April 1, 2014	First day of the 2nd quarter.
051714	ANYDTDTE6.	19860	May 17, 2014	
17052014	ANYDTDTE8.	19860	May 17, 2014	
170514	ANYDTDTE6.	20953	May 14, 2017	The date string can be interpreted as YMD or DMY. YMD is used because the DATESTYLE in effect for this example is MDY.
17MAY2014:15:12:06	ANYDTDTE18.	19860	May 17, 2014	
15:12:06	ANYDTDTE8.	0	January 1, 1960	Time value, date is considered to be January 1, 1960.
2014137	ANYDTDTE7.	19860	May 17, 2014	
MAY2014	ANYDTDTE7.	19844	May 1, 2014	No day is given; day is set to first day of month.
17MAY2014	ANYDTDTE9.	19860	May 17, 2014	
May 17, 2014	ANYDTDTE12.	19860	May 17, 2014	
14-05	ANYDTDE9.	19844	May 1, 2014	Two-digit year uses YEARCUTOFF= option. No day is given; day is set to first day of month.
2014-05	ANYDTDTE7.	19844	May 1, 2014	No day is given; day is set to first day of month.
05-2014	ANYDTDTE7.	19844	May 1, 2014	No day is given; day is set to first day of month.
05-17-2014 3:12:06 PM	ANYDTDTE21.	19860	May 17, 2014	
2014-05-17-15:12:06	ANYDTDTE19.	19860	May 17, 2014	

ANYDTDTM*w*.

ANYDTDTM*w*. will translate data that can be read with the following informats: DATE, DATETIME, DDMMYY, JULIAN, MDYAMPM, MMDDYY, MMxYY, MONYY, TIME, YMDDTTM, YYMMDD, YYMMn, or YYQ into SAS datetime values. This informat can also work with the following standard date formats: "February 25, 2012," "08/2014," or "2015-05." *w* can range from 1 to 32, and the default width is 19.

This informat will parse time values from input strings based on colons and periods. If there is one colon (for example, 15:12), the first two digits are translated as hours and the last two as minutes. If there are two colons (for example, 15:12:06), the digits preceding the first colon are translated as hours, the next set of two digits as minutes, and the last two as seconds. If there is a single decimal point and one or more colons (for example, 12:06.5), the value following the decimal point is translated as a decimal fraction of seconds, and the value preceding the decimal point is considered to be seconds. A single colon is therefore interpreted as *minutes:seconds.fractional seconds.* In order to account for hours, the correct format would be *hours:minutes:seconds.fractional seconds*.

If there are multiple decimal points (for example, 15.12.06), then they are considered to be delimiters for date values and thus are not translated as time values.

This informat extracts datetime values from a string. If only a time value is given, *the date is assumed to be January 1, 1960*. If only a date value is given, *the time is assumed to be midnight (0:00).* The following table uses the same input data as is used in the ANYDTDTE. informat example above.

Characters Read	**Informat**	**Resulting SAS Datetime Value**	**Formatted Datetime Using DATETIME22.**
05172014	ANYDTDTM.	1715904000	17MAY2014:00:00:00
20140517	ANYDTDTM.	1715904000	17MAY2014:00:00:00
2014Q2	ANYDTDTM.	1711929600	01APR2014:00:00:00
051714	ANYDTDTM.	1715904000	17MAY2014:00:00:00
17052014	ANYDTDTM.	1715904000	17MAY2014:00:00:00
170514	ANYDTDTM.	1810339200	14MAY2017:00:00:00
17MAY2014:15:12:06	ANYDTDTM.	1715958726	17MAY2014:15:12:06
15:12:06	ANYDTDTM.	54726	01JAN1960:15:12:06
2014137	ANYDTDTM.	1715904000	17MAY2014:00:00:00
MAY2014	ANYDTDTM.	1714521600	01MAY2014:00:00:00
17MAY2014	ANYDTDTM.	1715904000	17MAY2014:00:00:00
May 17, 2014	ANYDTDTM.	1715904000	17MAY2014:00:00:00

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME22.
2014-05	ANYDTDTM.	1714521600	01MAY2014:00:00:00
05-2014	ANYDTDTM.	1714521600	01MAY2014:00:00:00
14-05	ANYDTDTM.	1714521600	01MAY2014:00:00:00
05-17-2014 3:12:06 PM	ANYDTDTM.	1715958726	17MAY2014:15:12:06
2014-05-17-15:12:06	ANYDTDTM21.	1715958726	17MAY2014:15:12:06

ANYDTTME*w*.

ANYDTTME*w*. will translate data that can be read with the following informats: DATE, DATETIME, DDMMYY, JULIAN, MDYAMPM, MMDDYY, MMxYY, MONYY, TIME, YMDDTTM, YYMMDD, YYMMn, or YYQ into SAS time values. This informat can also work with the following standard date formats: "February 25, 2012," "08/2014," or "2015-05." *w* can range from 1 to 32, and the default width is 8. ANYDTTMEw. can extract time values from a datetime value. However, if only a date value is given, the time is assumed to be 12:00 a.m.

Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME8.	Comments
17MAY2014:15:12:06	ANYDTTME18.	54726	15:12:06	Extracted time from datetime string.
15:12:06	ANYDTTME.	54726	15:12:06	
05-17-2014 3:12:06 PM	ANYDTTME21.	54726	15:12:06	Extracted time from datetime string, processed AM/PM correctly.
2014-05-17-15:12:06	ANYDTTME18.	54726	15:12:06	Extracted time from datetime string.
08.05	ANYDTTME.	.	.	Times are delimited with colons (:).
08:05	ANYDTTME.	29100	8:05:00	
08:05:05	ANYDTTME.	29105	8:05:05	
17:15	ANYDTTME.	62100	17:15:00	
2:45 PM	ANYDTTME.	53100	14:45:00	

Characters Read	Informat	Resulting SAS Time Value	Formatted Time Using TIME8.	Comments
6 AM	ANYDTTME.	.	.	Need a delimiter to provide context for translation.
6:00 AM	ANYDTTME.	21600	6:00:00	
27:36:58	ANYDTTME.	99418	27:36:58	

3.4.5 So Why Not Just Use the "ANYDATE" Series of Informats?

It is tempting to automatically use ANYDTDTE., ANYDTTM., or ANYDTTME. to process strings representing dates, datetimes, and times. You do not have to worry about the formatting of the input string, and SAS will make sense out of it. The "ANYDATE" informats do have limitations, and the rules that SAS follows might not be the rules that you need to apply.

First, there is the matter of the additional processing needed. These informats go through a decision tree to determine how to translate every single value encountered; therefore, the amount of additional processing will increase with the number of times each "ANYDATE" informat has to be used. This would have a negligible impact on a small amount of data, but if you need to use them on big data, you might want to consider standardizing the representation of your date, time, and datetime text values beforehand and using the corresponding informat. Second, it is entirely possible that you would want to consider a nonstandard value erroneous and don't want SAS to decide what to do with it without you being able to inspect it first. Third, exceptions can occur, even when the use of an ANYDATE informat is warranted. While it is always a good idea to check all data that you are converting to SAS from another source and especially when you are converting dates, times, and datetime values, it is *critical* if you are using the ANYDT. informats.

Finally, it is important to note that the ANYDATE series of formats are designed to handle the issue of varying ways of representing dates, but they, like all other date, time, and datetime informats, are not designed to handle dirty data. Missing separators, misspelled words, and missing date components are a few of the major data problems that are encountered when processing dates, times, and datetimes from sources other than SAS. Unless you specifically provide code to fix those problems on the fly or can ensure that your dates are clean before processing them, you will wind up with missing date, time, or datetime values in your data set. SAS programmers have written a great deal of code to handle this specific issue; King and Fleming show one such approach in the 2011 SAS Global Forum proceedings (see http://support.sas.com/resources/papers/proceedings11/117-2011.pdf). Example 3.7 is a skeleton of an approach using a user-written function and the ANYDATE. informat.

Example 3.7: A Simple Function to Handle Dates with Missing Separators

```
LIBNAME control "c:\book\2ndEd\examples";
ODS ESCAPECHAR='~';
OPTIONS CMPLIB=(control.functions);

PROC FCMP OUTLIB=control.functions.dates;

FUNCTION makedate(cdate $)$;
  LENGTH dtstr $42
         supr   $16;
  supr=' ';
  * assume that date is in MMDDYY form, but check;
  * missing dates often have a form of __/__/____;
  IF SUBSTR(cdate,7,4) NOT IN (' ','____')  THEN DO;
     * assume year is good;
     yy=INPUT(SUBSTR(cdate,7,4),4.);
     m=SUBSTR(cdate,1,2);
     IF '01' <= m <= '12' THEN DO;
        mm=INPUT(m,2.);
       * Month is good chk day;
       * Use INTNX to find the maximum number of days in current
month;
       d=SUBSTR(cdate,4,2);
       IF '01' <= d <= PUT(DAY(INTNX('MONTH',MDY(mm,1,yy),0,'e')
),z2.)
          THEN dd=INPUT(d,2.);
       ELSE DO;
         * Day is bad for this month;
         * Fix by dividing number of days in month by 2;
         dd = CEIL(DAY(INTNX('MONTH',mdy(mm,1,yy),0,'e'))/2);
         supr = '~{super Day}';
       END;
     END;
     ELSE DO;
        * Month is bad - reset month and day;
        mm=6;
        dd=31;
        supr = '~{super Month}';
    END;
   * Create the date value;
     date=MDY(mm,dd,yy);
     dtstr = CATT(PUT(date,MMDDDYY10.),supr);
  END;
  ELSE DO;
    * yr is bad;
     dtstr = 'Unknown~{super Y}';
  END;
  RETURN(dtstr);
ENDSUB;
RUN;
QUIT;
```

```
PROC FORMAT;
   VALUE $mkdt
      OTHER=[makedate()];
RUN;
```

The code in Example 3.7 created a function to be used as a format for our dates, which will impute a replacement date string when provided with a date string that may or may not have missing date components. Note that this is a character format because it has to treat the incoming date data as character strings. If you try to read date strings with missing date components as a SAS date, you will get a missing value.

Now let's read in some sample data. The data set RPTDATES contains production error tracking data from a legacy system. Unfortunately, some of the date data was corrupted, so there are missing and incorrect days, months, and years.

```
DATA rptdates;
INPUT recno @4 rptdt $10. errs;
DATALINES;
1  13/15/2013 374
2  09/08/2013 3
3  02/30/2013 32
4  __/15/2013 15
5  04/27/2013 195
6  05/__/2013 17
7  07/08/____ 20
8  06/04/013 5
9  08/32/2013 4
10 11//2013   6
11 01/23/2031 11
;;;;
RUN;
```

But now that we've created a function that is to be used as a format, let's use it. Below is a data set with some dates and a count of machine faults occurring on that date. As you can see, the dates have not been entered very well, with missing components (rows 3, 5, and 6) and typographical errors (rows 7 and 9). We're going to use the format $MKDT to impute the dates we can, and annotate each date that was imputed in the output string.

Figure 3.1: RPTDATES Data Set with Invalid Dates

VIEWTABLE: Work.Rptdates

	recno	rptdt	errs
1	1	13/15/2013	374
2	2	09/08/2013	3
3	3	02/30/2013	32
4	4	__/15/2013	15
5	5	04/27/2013	195
6	6	05/__/2013	17
7	7	07/08/____	20
8	8	06/04/013	5
9	9	08/32/2013	4
10	10	11//2013	6
11	11	01/23/2031	11

Now we will use a PROC REPORT to display the table and format our bad data. In line 2, we alias the original date string as a second column in order to show it with and without the format that we created. We leave the original date string in the variable **rptdt** and in line 4 we apply the format that we created to the values in the aliased column **dt**.

```
1.  PROC REPORT DATA=rptdates NOWD SPLIT='\';
2.  COLUMN rptdt rptdt=dt errs;
3.  DEFINE rptdt / DISPLAY "Original\Date from\Error Log";
4.  DEFINE dt / 'Imputed\Date' f=$mkdt36.;
5.  DEFINE errs / "Errors\Reported";
6.  RUN;
```

The result of the PROC REPORT is below. The "Imputed Date" column is nothing more than the result of displaying the original date string using the $MKDT. format.

Output

Original Date from Error Log	Imputed Date	Errors Reported
13/15/2013	06/30/2013 Month	374

Original Date from Error Log	Imputed Date	Errors Reported
09/08/2013	09/08/2013	3
02/30/2013	02/14/2013[Day]	32
__/15/2013	06/30/2013[Month]	15
04/27/2013	04/27/2013	195
05/__/2013	05/16/2013[Day]	17
07/08/____	Unknown[Y]	20
06/04/013	Unknown[Y]	5
08/32/2013	08/16/2013[Day]	4
11//2013	11/15/2013[Day]	6
01/23/2031	Unknown[Y]	11

This capacity to use PROC FCMP to create functions that process character strings and then use the function you create in a custom informat (or format, for that matter) gives you the ability to create custom date informats. With a little creativity on your part, you can use this method to solve some of the more difficult date processing challenges that you may encounter.

Chapter 4: ISO 8601 Dates, Times, Datetimes, Durations, and Functions

4.1 What Is ISO 8601? 91
4.2 ISO 8601 Formats 92
4.3 ISO 8601 Informats 103
4.4 Time Zone Functions 111
4.5 ISO 8601 Durations and Intervals 116
4.6 Conclusion 136

4.1 What Is ISO 8601?

ISO 8601 is the name of an internationally accepted way of describing dates and times using numbers and is one of numerous international standards maintained by the International Standards organization (ISO). While using numbers eliminates the need for translating month names, the standard defines the order of date and time components in a date string, removing the confusion of whether 01-11-05 means January 11, 2005, November 1 2005, or November 5, 2001. This

facilitates the exchange of data, especially between international parties. The complete standard (available from http://www.iso.org/iso/home.html) covers the following:

- Date
- Time of day
- Coordinated universal time (UTC)
- Local time with offset to UTC
- Date and time
- Time intervals
- Recurring time intervals

This standard has been adopted by the Clinical Data Interchange Standards Consortium (CDISC) for the representation of its date and time data, so those in the pharmaceutical industry are particularly familiar with it.

SAS has built-in formats specifically designed to display its date, time, and datetime values in the way described by the standard. There is also a complete set of informats designed to interpret strings that conform to the standard so that they can easily be converted to SAS date, time, and datetime values and therefore used in calculations and stored in SAS data sets. One major difference between the ISO formats and informats and their standard counterparts is that the ISO standard allows for missing components. For example, if you have a date value where you only know the month and year, it can be explicitly represented in an ISO-standard date string. The standard SAS date formats can only represent a complete SAS date value. Even the standard formats that only display month and year use complete SAS date values, but the days are not part of the display. The ISO informats are a little more intricate because of this feature of the standard; some of the informats perform a default substitution for any missing components in order to create a valid SAS date, time, or datetime value. By comparison, the standard SAS date, time, and datetime informats will return a missing value if you present them with a string that has a missing component. Therefore, while it might seem that some of the formats and informats that we described in Chapters 2 and 3 will produce results according to the standard, it is much more reliable to use the formats and informats described in this chapter when you are working with ISO standard dates and times.

As with standard SAS formats, it is critical that you only use date formats and informats for date values, time formats and informats with time values, and datetime formats and informats with datetime values.

4.2 ISO 8601 Formats

All the ISO 8601 formats are left-justified, as opposed to the majority of standard SAS date, time, and datetime formats, which are right-justified because they represent numeric values. The formats in the following descriptions are grouped by date, time, and datetime. Each format has a basic (format name starts with "B") and an extended (format name starts with "E") version. The extended

version of the ISO formats uses delimiters between components, while the basic versions do not. All the following examples will present the basic version of the format, followed by the extended version of the same format with the same SAS value, so you can see the difference. It is important to note that SAS continues to develop formats and informats, so it is always a good idea to check the documentation that came with your release of SAS, or the online documentation at support.sas.com for any additional formats and/or informats.

4.2.1 ISO Date Formats

B8601DA*w*.

B8601DA*w*. writes date values in the ISO 8601 basic date notation *yyyyMMdd*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, and *dd* is the zero-padded numerical day of the month. It is left-justified within the field. *w* can be from 8 to 10, and the default width is 10. The following table shows the result when the date value is 19920, which corresponds to July 16, 2014.

Format Name	Result	Comment
B8601DA8.	20140716	
B8601DA9.	20140716	
B8601DA10.	20140716	

E8601DA*w*.

E8601DA*w*. writes date values in the ISO 8601 extended date notation *yyyy-MM-dd*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, and *dd* is the zero-padded numerical day of the month. It is left-justified. *w* is always 10. The following table shows the result when the date value is 19920, which corresponds to July 16, 2014.

Format Name	Result	Comment
E8601DA10.	2014-07-16	

4.2.2 ISO Time Formats

You can determine which of the ISO 8601 time formats you want to use by answering two questions. First, does the SAS time value that you are going to display represent UTC or local time? If it represents UTC, then you will use either the *8601TX. or the *8601TZ. format, and you have to determine whether you want to display UTC as its local time equivalent (based on the TIMEZONE= option setting). In that case, you would use the *8601TX. format. If you want to display the UTC time value as it is stored, use the *8601TZ. format. If the SAS time value represents local time, then use the *8601LZ. format. The TIMEZONE= option has no effect because the format understands that the time is the local time.

B8601TM*w.d*

B8601TM*w.d* writes time values in the ISO 8601 basic time notation *hhmmssffffff*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. Note that there are no delimiters in the ISO basic time string, which might make the result difficult to read. *w* can be from 6 to 15, with a default width of 8. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table shows the result when the time value is 61479.468, which corresponds to the time 5:04:39 p.m. (17:04:39.468):

Format Name	Result	Comment
B8601TM6.	170439	
B8601TM8.	170439	
B8601TM10.2.	17043947	
B8601TM12.	170439	
B8601TM15.3.	170439468	

E8601TM*w.d*

E8601TM*w.d* writes time values in the ISO 8601 extended time notation *hh:mm:ss.ffffff*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. Colons (:) delimit hours, minutes, and seconds, while a decimal point delimits the decimal portion of seconds. *w* can be from 8 to 15, with a default width of 8. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table shows the result when the time value is 61479.468, which corresponds to the time 5:04:39 p.m. (17:04:39.468):

Format Name	Result	Comment
E8601TM6.		
E8601TM8.	17:04:39	
E8601TM10.2.	17:04:39.5	Note the rounding of the fractional seconds.
E8601TM12.	17:04:39	
E8601TM15.3.	17:04:39.468	

B8601TX*w.d*

This format is available as of SAS version 9.4. B8601TX*w.d* adjusts a Coordinated Universal Time (UTC) value to the user's local time. Since it is intended for clock times only, values greater than 86400 or less than 0 will show asterisks. The format writes the adjusted local time in the ISO 8601 basic time notation *hhmmssffffff*+|−*hhmm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* represents decimal fraction of seconds, which is then followed by the time offset. The time offset is a plus (for time zones east of

the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hhmm*, all without delimiters, as this is a basic ISO format. The offset is calculated from the time at the zero meridian in Greenwich, England, using the TIMEZONE= system option. If TIMEZONE is not set, then the user local time is based on the system clock. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. You will need to ensure that you have a large enough width specification, or else the time zone offset could be truncated or dropped completely. The value is left-justified. The following table shows the result when the time value is 37050, which corresponds to the time 10:17:30 a.m. Greenwich Mean Time. As you can see, the value displayed changes with the time zone.

Format Name	Result	OPTIONS TIMEZONE=
B8601TX9.	144730+04	Asia/Kabul
B8601TX12.	171730+0700	Asia/Omsk
B8601TX14.	052500-0500	America/Winnipeg
B8601TX16.	221730+1200	Pacific/Fiji

E8601TX*w.d*

This format is available as of SAS version 9.4. E8601TX*w.d* adjusts a Coordinated Universal Time (UTC) value to the user's local time. Since it is intended for clock times only, values greater than 86400 or less than 0 will show asterisks. The format writes the local time in the ISO 8601 extended time notation *hh:mm:ss.ffffff+/-hh:mm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* represents decimal fraction of seconds, which is then followed by the time offset. The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hh:mm*. The offset is calculated from the time at the zero meridian in Greenwich, England, using the TIMEZONE= system option. If TIMEZONE is not set, then the user local time is based on the system clock. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table shows the result when the time value is 37050, which corresponds to the time 10:17:30 a.m. Greenwich Mean Time. As you can see, the value displayed changes with the time zone.

Format Name	Result	OPTIONS TIMEZONE=	Comment
E8601TX9.	12:17:30	Africa/Harare	Not enough space to display the offset, but the correct local time is displayed.
E8601TX.	12:17:30+02:00	Africa/Harare	Using the default length corrects the above problem.
E8601TX12.	15:47:30+05	Asia/Calcutta	
E8601TX14.	18:17:30+08:00	Asia/Manila	
E8601TX16.	12:17:30+02:00	Europe/Copenhagen	

B8601LZ*w.d*

This format is available as of SAS version 9.4. B8601LZ*w.d* displays SAS time values without any adjustments using the ISO 8601 basic time notation *hhmmssffffff*+|−*hhmm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* represents decimal fraction of seconds, which is then followed by the time offset from Coordinated Universal Time (UTC). The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hhmm*, all without delimiters, as this is a basic ISO format. The offset is calculated based on the time zone at the local SAS session It is not affected by the TIMEZONE= system option. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The main difference between this format and the B8601TX. format is that the SAS time value is considered to be local time, not Greenwich Mean Time. For easier comparison, we will use the same SAS time value as with the *8601TX examples. The following table shows the result when the time value is 37050, which corresponds to the time 10:17:30 a.m. Central Daylight time.

Format Name	Result	Comment
B8601LZ9.	101730Z	*Z* is displayed in place of the offset because there is not enough space for the offset.
B8601LZ12.	101730-0500	
B8601LZ14.	101730-0500	
B8601LZ16.	101730-0500	

E8601LZ*w.d*

This format is available as of SAS version 9.4. E8601LZ*w.d* displays SAS time values without any adjustments using the ISO 8601 extended time notation *hh:mm:ss.ffffff+/-hh:mm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* represents decimal fraction of seconds, which is then followed by the time offset from Coordinated Universal Time (UTC). The time offset is a plus (for time zones east of the zero

meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hh:mm*. The offset is calculated based on the time zone at the local SAS session. It is not affected by the TIMEZONE= system option. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The main difference between this format and the E8601TX. format is that the SAS time value is considered to be local time, not Greenwich Mean Time. For easier comparison, we will use the same SAS time value as with the *8601TX examples. The following table shows the result when the time value is 37050, which corresponds to the time 10:17:30 a.m. Central Daylight time.

Format Name	Result	Comment
E8601LZ9.	10:17:30Z	*Z* is displayed in place of the offset because there is not enough space for the offset.
E8601LZ12.	10:17:30Z	*Z* is displayed in place of the offset because there is not enough space for the offset.
E8601LZ14.	10:17:30-05:00	
E8601LZ16.	10:17:30-05:00	

B8601TZ*w.d*

This format is available as of SAS version 9.4. B8601TZ*w.d* displays a SAS time value as a Coordinated Universal Time (UTC) value. Since it is intended for clock times only, values greater than 86400 or less than 0 will show asterisks. The format writes the UTC time in the ISO 8601 basic time notation *hhmmssffffff*+|−*hhmm* where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* represents decimal fraction of seconds, which is then followed by the time offset. The time offset is always going to be displayed as +0000, because the resulting time display is the time at the zero meridian in Greenwich, England. This format is not affected by the TIMEZONE= system option. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table shows the result when the time value is 37050, which corresponds to the time 10:17:30 a.m. Greenwich Mean Time.

Format Name	Result	Comment
B8601TX9.	101730Z	*Z* is displayed in place of the offset because there is not enough space for the offset.
B8601TX12.	101730+0000	
B8601TX14.	101730+0000	
B8601TX16.	101730+0000	

E8601TZ*w.d*

This format is available as of SAS version 9.4. E8601TZ*w.d* displays a SAS time value as a Coordinated Universal Time (UTC) value. Since it is intended for clock times only, values greater than 86400 or less than 0 will show asterisks. The format writes the UTC time in the ISO 8601 extended time notation *hh:mm:ss.ffffff+/-hh:mm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* represents decimal fraction of seconds, which is then followed by the time offset. The time offset is always going to be displayed as +00:00, because the resulting time display is the time at the zero meridian in Greenwich, England. This format is not affected by the TIMEZONE= system option. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table shows the result when the time value is 37050, which corresponds to the time 10:17:30 a.m. Greenwich Mean Time.

Format Name	Result	Comment
E8601TX9.	10:17:30Z	*Z* is displayed in place of the offset because there is not enough space for the offset.
E8601TX.	10:17:30+00:00	Using the default length corrects the above problem.
E8601TX12.	10:17:30Z	
E8601TX14.	10:17:30+00:00	
E8601TX16.	10:17:30+00:00	

4.2.3 ISO Datetime Formats

B8601DN*w*.

B8601DN*w*. writes dates from datetime values in the ISO 8601 basic date notation *yyyyMMdd*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, and *dd* is the zero-padded numerical day of the month. It is left-justified within the field. *w* can be from 8 to 10, and the default width is 10. Note that the basic date notation has no delimiters. The datetime value used in this example is 1664263800, which corresponds to 7:30:00 a.m. on Wednesday, September 26, 2012.

Format Name	Result	Comment
B8601DN8.	20120926	
B8601DN9.	20120926	
B8601DN10.	20120926	

E8601DN*w*.

E8601DN*w*. writes dates from datetime values in the ISO 8601 extended date notation *yyyy-MM-dd*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, and *dd* is the zero-padded numerical day of the month. It is left-justified. *w* is always equal to 10, and the default width is 10. The datetime value used in this example is 1664263800, which corresponds to 7:30:00 a.m. on Wednesday, September 26, 2012.

Format Name	Result	Comment
E8601DN10.	2012-09-26	

B8601DT*w.d*

B8601DT*w.d* writes datetime values in the ISO 8601 basic datetime notation *yyyyMMddThhmmssffffff*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. *T* is the ISO

8601 delimiter for time. Note that there are no other delimiters in the ISO basic datetime string, including the decimal fractions, which might make the result difficult to read. *w* can be from 15 to 26, with a default width of 19. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The datetime value used in this example is 1686408430.44, which corresponds to 2:47:10.44 p.m. on Sunday, June 9, 2013.

Format Name	Result	Comment
B8601DT15.2.	20130609T144710	Not enough width to display fractional seconds.
B8601DT19.	20130609T144710	
B8601DT19.2.	20130609T14471044	
B8601DT24.1.	20130609T1447104	
B8601DT26.	20130609T144710	

E8601DT*w.d*

E8601DT*w.d* writes datetime values in the ISO 8601 extended datetime notation *yyyy-MM-ddThh:mm:ss.ffffff*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. *T* is the ISO 8601 delimiter for time. Colons (:) delimit hours, minutes, and seconds, while a decimal point delimits the decimal portion of seconds. *w* can be from 19 to 26, with a default width of 19. *d* can be from 0 to 6, with a default of 0.The value is left-justified. The datetime value used in this example is 1686408430.44, which corresponds to 2:47:10.44 p.m. on Sunday, June 9, 2013.

Format Name	Result	Comment
E8601DT19.	2013-06-09T14:47:10	
E8601DT19.2.	2013-06-09T14:47:10	Width not large enough to display fractional seconds.
E8601DT22.1.	2013-06-09T14:47:10.4	
E8601DT24.	2013-06-09T14:47:10	
E8601DT24.2.	2013-06-09T14:47:10.44	

B8601DX*w.d*

This format is available as of SAS version 9.4. B8601DX*w.d* adjusts a Coordinated Universal Time (UTC) datetime value to the user's local date and time. The format writes the adjusted local time in the ISO 8601 basic datetime and time zone notation *yyyyMMddThhmmssffffff*+|−*hhmm*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. Note that there are no delimiters in the ISO basic datetime string including the +/- of the

time offset. The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hhmm*, all without delimiters. The offset is calculated from the time at the zero meridian in Greenwich, England, using the TIMEZONE= system option. If TIMEZONE is not set, then the user local time is based on the system clock. *w* can be from 20 to 35, with a default width of 26. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table uses the value 1763371185, which corresponds to 9:19 a.m. on November 17, 2015, Greenwich Mean Time. As you can see, the display does not change depending on the width specification, because the minimum length of 20 can accommodate the entire width of the output.

Format Name	Result	OPTIONS TIMEZONE=
B8601DX20.	20151117T111945+0200	Africa/Cairo
B8601DX22.	20151117T031945-0600	America/Cancun
B8601DX24.	20151117T041945-0500	America/Indianapolis
B8601DX26.	20151117T171945+0800	Asia/Hong_Kong
B8601DX28.	20151117T194945+1030	Australia/Adelaide

E8601DX*w.d*

This format is available as of SAS version 9.4. E8601DX*w.d* adjusts a Coordinated Universal Time (UTC) datetime value to the user's local date and time. The format writes the adjusted local datetime in the ISO 8601 extended datetime and time zone notation *yyyy-MM-ddThhmmss.ffffff*+/-*hh:mm*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hh:mm*. The offset is calculated from the time at the zero meridian in Greenwich, England, using the TIMEZONE= system option. If TIMEZONE is not set, then the user local time is based on the system clock. *w* can be from 20 to 35, with a default width of 26. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The following table uses the value 1763371185, which corresponds to 9:19 a.m. on November 17, 2015, Greenwich Mean Time.

Format Name	Result	OPTIONS TIMEZONE=	Comment
E8601DX20.	2015-11-17T02:19:45	America/Edmonton	Not enough space for the time offset.
E8601DX22.	2015-11-17T05:19:45-04	America/Halifax	Not enough space for minutes of the time offset. As some offsets aren't full hours, this can give you an incorrect result.
E8601DX26.	2015-11-17T03:19:45-06:00	America/Mexico_City	
E8601DX28.	2015-11-17T19:49:45+10:30	Australia/Adelaide	

B8601DZ*w.d*

B8601DZ*w.d* displays a SAS datetime value based on the zero meridian Coordinated Universal Time (UTC) in the ISO 8601 basic datetime and time zone notation *yyyyMMddThhmmssffffff*+|−*hhmm*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. The time offset is always going to be displayed as +0000, because the resulting time display is the time at the zero meridian in Greenwich, England. Therefore, this format is not affected by the TIMEZONE= system option. While the SAS documentation says that *w* can be from 16 to 35, with a default width of 26, you will get a warning and a missing value if you use a format width of less than 20. *d* can be from 0 to 6, with a default of 0. The value is left-justified. The datetime value used in this example is 1730398875, which corresponds to 6:21:15 p.m. on October 31, 2014, Greenwich Mean Time.

Format Name	Result	Comment
B8601DZ20.	20141031T182115+0000	Minimum format width to obtain a result.
B8601DZ26.	20141031T182115+0000	

E8601DZ*w.d*

E8601DZ*w.d* displays a SAS datetime value based on the zero meridian Coordinated Universal Time (UTC) in the ISO 8601 extended datetime and time zone notation *yyyy-MM-ddThhmmss.ffffff+/-hh:mm*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. The time offset is always going to be displayed as +00:00, because the resulting time display is the time at the zero meridian in Greenwich, England. Therefore, this format is not affected by the TIMEZONE= system option. *w* can be from 20 to 35,

with a default width of 26, and the value is left-justified. *d* can be from 0 to 6, with a default of 0. The datetime value used in this example is 1730398875, which corresponds to 6:21:15 p.m. on October 31, 2014, Greenwich Mean Time.

Format Name	Result	Comment
E8601DZ20.	2014-10-31T18:21:15Z	*Z* is used to indicate that the format width isn't wide enough to accommodate the time zone notation.
E8601DZ22.	2014-10-31T18:21:15Z	*Z* is used to indicate that the format width isn't wide enough to accommodate the time zone notation
E8601DZ26.	2014-10-31T18:21:15+00:00	
E8601DZ28.	2014-10-31T18:21:15+00:00	

4.3 ISO 8601 Informats

ISO 8601 informats are designed to process strings of characters that represent ISO 8601 dates, times, and datetimes and, when used with the INPUT statement or INPUT() or INPUTN() functions, will produce a SAS date, time, or datetime value when the string is read with the selected informat.

4.3.1 ISO Date Informats

B8601DA*w*.

B8601DA*w*. will read date values in both the ISO 8601 basic date notation *yyyyMMdd* and the ISO 8601 extended date notation *yyyy-MM-dd*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, and *dd* is the zero-padded numerical day of the month. *w* is always 10, and the default is 10. It is important to note that if either month or day is missing, SAS will use a value of 1 for the month and/or day to provide a SAS date value. However, this might not be what you want, as your situation might call for an algorithm to impute dates with missing months and/or days instead.

Characters Read	Informat	SAS Date Value	Formatted Value Using B8601DA. Format	Comment
20140504	B8601DA.	19847	20140504	
201405	B8601DA.	19844	20140501	Missing day, so date is set to first of the month.
2014	B8601DA.	19724	20140101	Missing month and day, so date is set to first of the year.

E8601DA*w*.

E8601DA*w*. reads date values in the ISO 8601 extended date notation *yyyy-MM-dd*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, and *dd* is the zero-padded numerical day of the month. This informat can only have a length of 10, as that is the only length for a date in ISO 8601 extended date notation. Unlike the parallel basic date notation informat, SAS will not substitute a value of 1 for either missing month or day, and you will get a missing value for your SAS date.

Characters Read	Informat	SAS Date Value	Formatted Value Using B8601DA. Format	Comment
2014-05-	E8601DA10.	19847	2014-05-04	
2014-05	E8601DA10.			Incomplete date yields missing value.
2014	E8601DA10.			Incomplete date yields missing value.

4.3.2 ISO Time Informats

B8601TM*w.d*

B8601TM*w.d* reads time values in the ISO 8601 basic time notation *hhmmssffffff*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. *w* can be from 6 to 15, with 8 as the default. Using a *w* of 6 will read time values with no decimal portion. *d* can be from 0 to 6, with a default of 0.

Characters Read	Informat	SAS Time Value	Formatted Value Using B8601TM. Format
144535	B8601TM8.	53135.0	144535
0630	B8601TM8.	23400.0	063000
1208455	B8601TM10.1	43725.5	120846

E8601TM*w.d*

E8601TM*w.d* reads time values in the ISO 8601 extended time notation *hh:mm:ss.ffffff*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. *w* can be from 8 to 15, with 8 as the default. Using a *w* of 8 will read time values with no decimal portion. *d* can be from 0 to 6, with a default of 0.

Characters Read	Informat	SAS Time Value	Formatted Value Using E8601TM. Format	Comment
10:17:45	E8601TM8.	37065.00	10:17:45.00	
18:05	E8601TM8.	65100.00	18:05:00.00	Only missing seconds are set to zero; any other missing components will result in a missing time value.
07:15:12.25	E8601TM12.2	26112.25	07:15:12.25	

B8601TZ*w.d*

B8601TZ*w.d* reads Coordinated Universal Time (UTC) time values using the ISO 8601 basic time notation *hhmmssffffff,*+|−*hhmm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents zero-padded seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time

represented as *hhmm*. The time zone offset might also be indicated by the letter *Z*, representing the zero meridian. The offset is calculated from the time at the zero meridian in Greenwich, England, and the resulting SAS time value will be the time at the zero meridian. *w* can be from 9 to 20, with a default width of 14, while *d* can be from 0 to 6, with a default of 0. Since this informat is intended to only read clock times, values resulting in times greater than 24:00:00 or less than 00:00:00 will be adjusted appropriately so that the result will be within the 24-hour clock.

Characters Read	Informat	SAS Time Value	Formatted Value Using B8601TZ. Format	Comment
175200+0000	B8601TZ14.	64320	175200+0000	
175200Z	B8601TZ9.	64320	175200+0000	If the time at the zero meridian is represented by a Z, you should use the format width of 9.
091520+0600	B8601TZ14.	11720	031520+0000	
210800-0500	B8601TZ14.	7680	020800+0000	Instead of 260800, the value is adjusted to 020800.

E8601TZ*w.d*

E8601TZ*w.d* reads Coordinated Universal Time (UTC) time values using the ISO 8601 extended time notation *hh:mm:ss.ffffff*+|−*hh:mm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents zero-padded seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hh:mm*. The time zone offset might also be indicated by the letter *Z*, representing the zero meridian. The offset is calculated from the time at the zero meridian in Greenwich, England, and the resulting SAS time value will be the time at the zero meridian. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. Since this informat is intended to only read clock times, values resulting in times greater than 24:00:00 or less than 00:00:00 will be adjusted appropriately so that the result will be within the 24-hour clock.

Characters Read	Informat	SAS Time Value	Formatted Value Using E8601TZ20.2 Format	Comment
17:52:00+00:00	E8601TZ14.	64320.00	17:52:00.00+00:00	
17:52:00Z	E8601TZ9.	64320.00	17:52:00.00+00:00	If the time at the zero meridian is represented by a *Z*, you should use the format width of 9.

Characters Read	Informat	SAS Time Value	Formatted Value Using E8601TZ20.2 Format	Comment
06:00:30.57+08:00	**E8601TZ18.2**	**79230.57**	**22:00:30.57+00:00**	**Adjusted to give the time value of 22:00:30.57 instead of -02:00:30.57.**
04:17:00-05:00	E8601TZ14.	33420.00	09:17:00.00+00:00	

You might wonder why a UTC value of 06:00:30.57+08:00 yields a time value of 22:00:30.57 at the zero median, and not 14:00:30.57. The +08:00 hour offset means that GMT plus 8 hours will give you the local time. Therefore, the conversion from local time to time at the zero median is local time minus the offset. 6-8=-2, which is then adjusted to 22 by adding 24 hours.

E8601LZ*w.d*

E8601LZ*w.d* reads Coordinated Universal Time (UTC) time values using the ISO 8601 extended time notation *hh:mm:ss.ffffff*+|−*hh:mm*, where *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents zero-padded seconds, and *ffffff* is decimal fractions of seconds, which is then followed by the time offset. The time offset is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hh:mm*. The time zone offset might also be indicated by the letter *Z*, representing the zero meridian. The offset is calculated from the time at the zero meridian in Greenwich, England, and the resulting SAS time value will be the time of the local SAS session, based on the system clock. It is not affected by the TIMEZONE= system option. *w* can be from 9 to 20, with a default width of 14. *d* can be from 0 to 6, with a default of 0. Since this informat is intended to only read clock times, values resulting in times greater than 24:00:00 or less than 00:00:00 will be adjusted appropriately so that the result will be within the 24-hour clock.

Characters Read	Informat	SAS Time Value	Formatted Value Using E8601LZ. Format	Comment
17:52:00+00:00	E8601LZ14.	64320.00	17:52:00.00-05:00	The time of the local SAS session is GMT-5.
17:52:00Z	E8601LZ9.	64320.00	17:52:00.00-05:00	If the time at the zero meridian is represented by a *Z*, you should use the format width of 9.
06:00:30.57+08:00	E8601LZ18.2	79230.57	22:00:30.57-05:00	
04:17:00-05:00	E8601LZ14.	33420.00	09:17:00.00-05:00	

4.3.3 ISO Datetime Informats

B8601CI*w.d*

B8601CI*w.d* reads IBM time values with a century marker of the form *cyyMMddhhmmss<fff>***,** where *c* represents the century digit. The century digit is calculated by subtracting 1900 from the current year, dividing by 100, and dropping the remainder. *yy* is the two-digit year from 00 to 99, *MM* represents the number of the month, and *dd* represents the day of the month. The time is represented by *hhmmss<fff>*, where *hh* indicates the hours, *mm* is minutes, *ss* is the number of seconds, and *fff* indicates thousandths of seconds. *w* ranges from 10 to 26, with a default value of 16, while *d* ranges from 0 to 6 for the fractional part of seconds. However, it is important to note that there are only 3 places of decimal precision.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME25.3.
11504231905	B8601CI16.	1745435100.0	23APR2015:19:05:00.000
0560928053505	B8601CI16.	-102795895.0	28SEP1956:05:35:05.000
1140630102416454	B8601CI19.3	1719743056.5	30JUN2014:10:24:16.454
2131216094500	B8601CI16.	4858479900.0	16DEC2113:09:45:00.000

B8601DJ*w.d*

B8601DJ*w.d* reads datetimes in standard Java date and time notation *yyyyMMddhhmmssffffff*, where *yyyy* is the four-digit year, *MM* represents the number of the month, *dd* represents the day of the month, and time is represented by *hhmmss<ffffff>*, where *hh* indicates the hours, *mm* is minutes, *ss* is the number of seconds, and *ffffff* indicates millionths of seconds. *w* ranges from 10 to 26, with a default value of 16, while *d* ranges from 0 to 6 for the fractional part of seconds. The following table gives examples of how to apply this informat to yield the SAS date value that corresponds to the text shown in each line.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME25.4
201607181108	B8601DJ16.	1784459280.0	18JUL2016:11:08:00.0000
20141123054509	B8601DJ16.	1732340709.0	23NOV2014:05:45:09.0000
201303070814433064	B8601DJ21.4	1678263283.3	07MAR2013:08:14:43.3064
201406241630254	B8601DJ16.1	1719246625.4	24JUN2014:16:30:25.4000

B8601DT*w.d*

B8601DT*w.d* reads datetime values in the ISO 8601 basic datetime notation *yyyyMMddThhmmssffffff*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. *T* is the ISO 8601 delimiter for time. Note that there are no other delimiters in the ISO basic datetime string, including the decimal fractions. *w* can be from 19 to 26, with a default width of 19. *d* can be from 0 to 6, with a default of 0. It is important to note that if either month or day is missing, SAS will use a value of 1 for the month and/or day to provide the date for the SAS datetime value, while setting the hours, minutes, and seconds to zero. However, this might not be what you want, as your situation might call for an algorithm to impute datetimes with missing months and/or days, and/or times instead.

Characters Read	Informat	Resulting SAS Datetime Value	Formatted Datetime Using DATETIME25.4
20141007T133008745	B8601DT19.3	1728307808.7	07OCT2014:13:30:08.7450
20150716T0859003315	B8601DT19.4	1752656340.3	16JUL2015:08:59:00.3315
20140331T1404	B8601DT19.	1711893840.0	31MAR2014:14:04:00.0000
20150903T06	B8601DT19.	1756879200.0	03SEP2015:06:00:00.0000
20140804	B8601DT19.	1722729600.0	04AUG2014:00:00:00.0000
201312	B8601DT19.	1701475200.0	01DEC2013:00:00:00.0000
2016	B8601DT19.	1767225600.0	01JAN2016:00:00:00.0000

E8601DT*w.d*

E8601DT*w.d* reads datetime values in the ISO 8601 extended datetime notation *yyyy-MM-ddThh:mm:ss.ffffff*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is decimal fractions of seconds. *T* is the ISO 8601 delimiter for time. *w* can be from 19 to 26, with a default width of 19. *d* can be from 0 to 6, with a default of 0. Unlike the parallel basic datetime informat, there is no default substitution for missing date and/or time components, so the result is a missing datetime value.

Characters Read	Informat	SAS Datetime Value	Formatted Value Using MDYAMPM21. Format	Comment
2014-10-07T13:30:08	E8601DT19.	1728307808	10/7/2014 1:30 PM	
2015-07-16T08:59:00	E8601DT19.	1752656340	7/16/2015 8:59 AM	

Characters Read	Informat	SAS Datetime Value	Formatted Value Using MDYAMPM21. Format	Comment
2014-03-31T14:04	E8601DT19.	1711893840	3/31/2014 2:04 PM	
2015-09-03T06	E8601DT19.			Incomplete time yields a missing value.
2014-08-04	E8601DT19.			No time provided. Therefore, datetime is missing.
2013-12	E8601DT19.			Partial date; datetime is missing.
2016	E8601DT19.			Partial date; datetime is missing.

B8601DZ*w.d*

B8601DZ*w.d* reads a SAS datetime value based on the zero meridian Coordinated Universal Time (UTC) in the ISO 8601 basic datetime and time zone notation *yyyyMMddThhmmssffffff*+|−*hhmm*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd* is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is the decimal fraction of seconds. This is followed by the time offset, which is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hhmm*. The time zone offset might also be indicated by the letter *Z*, representing the zero meridian. The offset is calculated from the time at the zero meridian in Greenwich, England, and the resulting SAS datetime value will be the datetime at the zero meridian. *w* can be from 20 to 35, with a default width of 26. *d* can be from 0 to 6, with a default of 0.

Characters Read	Informat	SAS Datetime Value	Formatted Value Using E8601DZ. Format
20150208T112705+0500	B8601DZ.	1738996025	2015-02-08T06:27:05.00+00:00
20150920T05045914-0400	B8601DZ26.2	1758359099.14	2015-09-20T09:04:59.14+00:00
20140511T211700Z	B8601DZ.	1715462220	2014-05-11T21:17:00.00+00:00
20140511T211700+0000	B8601DZ.	1715462220	2014-05-11T21:17:00.00+00:00

E8601DZ*w.d*

E8601DZ*w.d* reads a SAS datetime value based on the zero meridian Coordinated Universal Time (UTC) in the ISO 8601 extended datetime and time zone notation *yyyy-MM-ddThh:mm:ss.ffffff*+|−*hh:mm*, where *yyyy* is the four-digit year, *MM* is the zero-padded month, *dd*

is the zero-padded numerical day of the month, *hh* represents zero-padded hours, *mm* represents zero-padded minutes, *ss* represents seconds, and *ffffff* is the decimal fraction of seconds. This is followed by the time offset, which is a plus (for time zones east of the zero meridian) or a minus (for time zones west of the meridian), followed by the offset time represented as *hh:mm*. The time zone offset might also be indicated by the letter *Z*, representing the zero meridian. The offset is calculated from the time at the zero meridian in Greenwich, England, and the resulting SAS datetime value will be the datetime at the zero meridian. *w* can be from 20 to 35, with a default width of 26. *d* can be from 0 to 6, with a default of 0.

Characters Read	Informat	SAS Datetime Value	Formatted Value Using E8601DZ. Format
2015-02-08T11:27:05+05:00	E8601DZ26.	1738996025	2015-02-08T06:27:05.00+00:00
2015-09-20T05:04:59.14-04:00	E8601DZ29.2	1758359099.14	2015-09-20T09:04:59.14+00:00
2014-05-11T21:17:00Z	E8601DZ26.	1715462220	2014-05-11T21:17:00.00+00:00
2014-05-11T21:17:00+00:00	E8601DZ26.	1715462220	2014-05-11T21:17:00.00+00:00

4.4 Time Zone Functions

4.4.1 Introduction

There are several time zone functions available in SAS. They are provided as part of National Language Support (NLS), but these functions are covered in this section because this is where the formats and informats that provide for time zone offsets and support for UTC are detailed.

For all of the following functions, *time-zone-id* represents a SAS time-zone ID value. It is also an optional argument in these functions. If it is not provided, the function will use the current setting of the TIMEZONE= system option as the default. Before using these functions, you need to check the setting of your TIMEZONE= system option to make sure that it does have a value, or many of these functions will produce a missing result. This option can be locked by your SAS administrator.

4.4.2 The TIMEZONE= Option

Many of the ISO time and datetime formats use the TIMEZONE= system option, which is available as of SAS version 9.4. The TIMEZONE= system option allows you to set a time zone based on a geographical location. It is not set by default. This option affects the following:

- Times that are recorded in logs and events.
- Creation and modification time stamps on SAS data sets
- The DATE(), DATETIME(), TIME(), and TODAY() functions

- The time zone formats B8601DX*w.d*, E8601DX*w.d*, B8601LX*w.d*, E8601LX*w.d*, B8601TX*w.d*, E8601TX*w.d*, NLDATMZw., NLDATMTZw., and NLDATMWZw.
- The time zone functions described in Section 4.4.3.

The value of the option may be represented in one of two ways: It may be a three- or four-letter acronym that describes the time zone (for example, EST, for Eastern Standard Time), or a time-zone ID that specifies a region and an area, separated by a forward slash (/), such as "America/New_York." The time zone ID values are unique and compatible with Java time zone names, while the three- or four-letter abbreviations are not. Consider the abbreviation "CST", which stands for "China Standard Time," or "Cuba Standard Time," or "Central Standard Time." How does SAS choose what "CST" represents in your program? It uses the value of the LOCALE= system option to decide what the correct region and area should be. Of course, if you are running a SAS program in the United States, but are using the LOCALE= system option to produce output for another geographical region, this may cause incorrect timing. I would recommend using the time zone ID instead of the acronyms as a best practice. There are over 500 time zone ID values; consult the SAS documentation to find the time zone ID values you need.

4.4.3 List of Time Zone Functions

Note that all of these functions are only available beginning with SAS version 9.4.

TZONEID(time-zone-id)

TZONEID will return either the current time zone ID if *time-zone-id* is a valid value, or a blank when *time-zone-id* is missing or invalid. Note: Prior to SAS 9.4 TS1M2, supplying a *time-zone-id* for this function will cause an error.

Sample Function Call	OPTIONS TIMEZONE=	Result
TZONEID()		
TZONEID()	Africa/Addis_Ababa	AFRICA/ADDIS_ABABA
TZONEID()	Africa/Brazzaville	AFRICA/BRAZZAVILLE
TZONEID()	America/Montreal	AMERICA/MONTREAL

TZONENAME(time-zone-id,datetime-value)

TZONENAME returns the current time zone name based on the time zone ID and any daylight saving time rules. This function will return a blank if *time-zone-id* is missing, or the TIMEZONE= system option has no value. In the following example, the TIMEZONE option is set to 'America/Chicago.'

Sample Function Call	Result	Comment
TZONENAME()	CDT	Based on the current time zone and time of year, Central Daylight Time is returned.
TZONENAME('06JUN2014:08:00'dt)	CDT	Given the datetime value provided, Central Daylight Time is returned. Daylight saving time is in effect in June for the Central time zone.
TZONENAME('06JAN2015:08:00'dt)	CST	Given the datetime value provided, Central Standard Time is returned. Daylight saving time is NOT in effect in January for the Central time zone.

TZONEOFF(time-zone-id,datetime-value)

TZONEOFF returns the time zone offset from Coordinated Universal Time (UTC) based on time zone name, and standard or daylight saving time rules. If *time-zone-id* is blank or missing, the offset will be determined from the system clock. In the following example, the TIMEZONE option is set to 'Europe/Stockholm.'

Sample Function Call	Result	Comment
TZONEOFF()	2:00	Based on the current time zone and time of year, Central European Summer Time is in effect.
TZONEOFF('06JUN2014:08:00'dt)	2:00	Based on the datetime value provided, Central European Summer Time is in effect.
TZONEOFF('06JAN2015:08:00'dt)	1:00	According to the datetime provided, Central European Standard Time is in effect.

TZONES2U(datetime-value, time-zone-id)

TZONES2U converts a SAS datetime value to a Coordinated Universal Time (UTC) datetime value based on the time zone and daylight saving rules. If *time-zone-id* is blank or missing, the offset will be determined from the system clock. In the following example, the TIMEZONE option is set to 'America/Chicago.'

Sample Function Call	Result
TZONES2U('11NOV2014:12:00'dt)	11NOV2014:18:00:00
TZONES2U('11NOV2014:12:00'dt,'Pacific/Guam')	11NOV2014:02:00:00
TZONES2U('11NOV2014:12:00'dt,'Europe/Istanbul')	11NOV2014:10:00:00

TZONEDSTNAME(time-zone-id)

TZONEDSTNAME returns the daylight saving time abbreviation for the time zone ID. If *time-zone-id* is missing, and the TIMEZONE= option is not set, you will receive a "NOTE: Invalid argument to function TZONEDSTNAME()" message in the log. If the time zone provided does not follow daylight saving time, then the result will be missing. In the following example, the TIMEZONE option is set to 'America/Chicago.'

Sample Function Call	Result	Comments
TZONEDSTNAME()	CDT	
TZONEDSTNAME('Asia/Calcutta')		The Indian Time Zone does not observe daylight saving time, so the result is missing.
TZONEDSTNAME('America/Sao_Paulo')	BRST	

TZONEDSTOFF(time-zone-id)

TZONEDSTOFF returns the time zone offset value for the specified daylight saving time. If *time-zone-id* is missing, and the TIMEZONE= option is not set, you will receive a "NOTE: Invalid argument to function TZONEDSTOFF()" message in the log. In the following example, the TIMEZONE option is set to 'America/Chicago.'

Sample Function Call	Result	Comments
TZONEDSTOFF()	-5:00	
TZONEDSTOFF('Asia/Calcutta')	.	The Indian Time Zone does not observe daylight saving time, so the result is missing.
TZONEDSTOFF('America/Sao_Paulo')	-2:00	

TZONESTTNAME(time-zone-id)

TZONESTTNAME returns the standard time name for the time zone ID. This function differs from the TZONEDSTNAME() function in that it provides the standard time abbreviation as opposed to the daylight saving time abbreviation. If *time-zone-id* is missing, and the TIMEZONE= option is not set, you will receive a "NOTE: Invalid argument to function

TZONESTTNAME()" message in the log. In the following example, the TIMEZONE option is set to 'Pacific/Honolulu.'

Sample Function Call	Result
TZONESTTNAME()	HST
TZONESTTNAME('Australia/Sydney')	EST
TZONESTTNAME('Asia/Dubai')	GST

TZONESTTOFF(time-zone-id)

TZONESTTOFF returns the time zone offset value for the specified daylight saving time. If *time-zone-id* is missing, and the TIMEZONE= option is not set, you will receive a "NOTE: Invalid argument to function TZONESTTOFF()" message in the log. In the following example, the TIMEZONE option is set to 'Pacific/Honolulu.'

Sample Function Call	Result
TZONESTTOFF()	-10:00
TZONESTTOFF('Australia/Sydney')	10:00
TZONESTTOFF('Asia/Dubai')	4:00

TZONEU2S(UTC-datetime-value, time-zone-id)

TZONEU2S converts a Coordinated Universal Time (UTC) datetime value to a SAS datetime value. If *time-zone-id* is blank or missing, the offset will be determined from the system clock and according to daylight saving time rules in effect. In the following example, the TIMEZONE option is set to 'America/Chicago.'

Sample Function Call	Result
TZONEU2S('11NOV2014:12:00'dt)	11NOV2014:06:00:00
TZONEU2S('11NOV2014:12:00'dt,'Pacific/Guam')	11NOV2014:22:00:00
TZONEU2S('11NOV2014:12:00'dt,'Europe/Istanbul')	11NOV2014:14:00:00

4.5 ISO 8601 Durations and Intervals

The ISO 8601 standard also provides a way to describe the interval between two datetimes, or a period of time over which an event has occurred. An interval can be expressed by using the starting and ending datetimes, or a duration and a starting time. Durations can be expressed by providing the length of time in a common form. However, ISO 8601 durations and time intervals do not directly equate to a single point in time, while everything else in the SAS date and time facility represents one specific point in time. Nonetheless, SAS gives you the ability to handle durations and intervals so that you have access to all the functionality of the SAS date and time facility.

Unlike every other part of the SAS date and time facility, ISO 8601 durations and intervals are stored in character variables, but they are not stored as simple text. You cannot convert a datetime to a character string using the PUT() function and then concatenate it with a duration string or another datetime that has been converted to a character string. In order to use the SAS date and time capacity with ISO durations and intervals, you must first convert ISO duration and interval values to an internal representation. This is very much the same process that you have to undertake when you are working with dates and times as we understand them; you must turn them into values that SAS understands or take the values that SAS understands and translate them into the representation that we understand.

Therefore, SAS has formats and informats dedicated to translating this internal representation of ISO durations and intervals. The internal representation cannot be used without this translation. There is also one extremely important CALL routine that is used to effect the transformation between ISO durations and intervals to or from SAS date, time, and datetime values. First, we should talk about the forms of ISO durations and intervals.

4.5.1 ISO Duration and Interval Representations

ISO Duration Representations

ISO durations are represented in three ways. There is no preference or priority attached to any of the forms: Use the one that best fits your data. One form is **P**n**Y**n**M**n**DT**n**H**n**M**n**S**, where **P** is the indicator for period, and is preceded by a minus sign if the period represented is negative (the "starting" date or datetime comes after the "ending" date or datetime), **Y** is the indicator for years, **M** for months, and **D** for days. The **T** is the indicator for time and must be present if there is a time component to the duration string. **H** is the indicator for hours, the **M** to the right of the **T** indicator is for minutes, and the **S** is the indicator for seconds. *n* represents a number. This form of duration description is the same for both the basic and extended notation, as there are no delimiters necessary.

Another duration form is **P**n**W**, which is only used to describe a duration measured in weeks. *n* represents a number, and the **W** is the indicator for weeks. As with the previous form, when the starting date or datetime is after the "ending" date or datetime, the **P** is preceded by a minus sign. Again, since there are no delimiters in this form, the output using basic and extended notation is identical.

The last of the duration forms is **P**yyyyMMdd**T**hhmmssfff in basic notation, or **P**yyyy-MM**-**dd**T**hh**:**mm**:**ss.fff, in the extended notation, where *yyyy* is the number of years, *MM* is the number of months, *dd* is number of days, *hh* represents number of hours, *mm* represents minutes, and *ss* represents seconds in the period, with *fff* as decimal fractions of seconds up to the millisecond. **T** is the ISO 8601 delimiter for time. This duration form can also be represented as **P**yyyyMMdd or **P**yyyy-MM**-**dd if there is no time component. As with the other duration forms, if the "start" is later than the "end," the P will be preceded by a minus sign.

ISO Interval Representations

ISO intervals are represented by two datetimes that represent the start and end of the interval, or a duration and a datetime that represents either the start or end of the interval. The datetime/datetime form is yyyyMMdd**T**hhmmss/yyyyMMdd**T**hhmmss in the basic notation, or yyyy-MM-dd**T**hh**:**mm**:**ss/yyyy-MM**-**dd**T**hh**:**mm**:**ss in the extended notation. The slash or solidus (/) between the two datetimes is a required delimiter when using either notation.

In the basic notation, the duration/datetime form is: **P**n**Y**n**M**n**DT**n**H**n**M**n**S**/yyyyMMdd**T**hhmmssfff when the datetime represents the end of the interval, or yyyyMMdd**T**hhmmssfff/**P**n**Y**n**M**n**DT**n**H**n**M**n**S** when the datetime value represents the beginning of the interval. In extended notation, the forms are yyyy-MM-dd**T**hh**:**mm**:**ss.fff/**P**n**Y**n**M**n**DT**n**H**n**M**n**S** or **P**n**Y**n**M**n**DT**n**H**n**M**n**S/**yyyy-MM-dd**T**hh**:**mm**:**ss.fff. The duration segment is specified by **P**, the indicator for period, **Y** is the indicator for years, **M** for months, **D** for days. The **T** is the indicator for time, and must be present if there is a time component to the duration string. **H** is the indicator for hours, the **M** to the right of the **T** indicator is for minutes, and the **S** is the indicator for seconds. *n* represents a number.

4.5.2 ISO 8601 Duration and Interval Formats

The first thing you might notice about the ISO duration and interval formats is that, unlike any of the other date- and time-related formats in SAS, they are character formats, as indicated by the leading dollar sign ($). This is reasonable, since durations and intervals are stored as character values. Why do you need formats for character variables? SAS stores ISO durations and intervals in an internal form, so just as you need formats to convert between SAS date values and how we understand dates, you need these formats to perform the conversion from the stored form to the corresponding ISO 8601 representation. Here's an example that shows what the stored form of duration and interval values looks like, along with its formatted display. The example uses the CALL IS8601_CONVERT routine to create the internal representations, and we will discuss this routine, its syntax, and use in detail in section 4.5.4.

Example 4.1: Why Formats are Necessary with ISO Durations, Intervals, and Datetimes

```
1.  DATA isostore;
2.  SET isotest;
3.  LENGTH result1-result2 $ 32;
4.  CALL IS8601_CONVERT('dt/dt','du',dt1,dt2,result1);
5.  CALL IS8601_CONVERT("dt/dt",'intvl',dt1,dt2,result2);
```

```
6.  fmt_result1 = PUT(result1,$N8601E.);
7.  fmt_result2 = PUT(result2,$N8601E.);
8.  RUN;
```

The above code will take the datetime values dt1 and dt2 from a data set. The CALL IS8601_CONVERT in line 4 creates an ISO duration value in the variable **result1**, while the one in line 5 creates an ISO interval value in the variable **result2**. The two results are then formatted in new variables using the PUT function. The values in this data set are displayed below.

Start of period (dt1)	End of Period (dt2)	Duration Value Stored in Dataset (result1)	Formatted Duration Value (fmt_result1)
04MAR2014:10:23:23	28DEC2014:23:04:03	***FFFF924124040FFC***	P9M24DT12H40M40S

Interval Value Stored in Dataset (result2)	Formatted Interval Value (fmt_result2)
20143041023230012014C28230403001	2014-03-04T10:23:23.000/2014-12-28T23:04:03.000

Clearly, the stored duration and interval values (italicized in the above table) are not in the ISO 8601 format, so SAS provides the following formats to translate them into the desired representation.

$N8601B*w.d*

$N8601B*w*. writes duration, datetime, and interval values in the ISO 8601 basic notation. This format differs from the $N8601BA. format in that it displays durations in the form P*n*Y*n*M*n*DT*n*H*n*M*n*S instead of P*yyyymmdd*T*hhmmss*. It is left-justified, and *w* can be from 1 to 200, with a default width of 50. In order to display datetime and duration values correctly, the width must be at least 16, while to display interval values correctly, the minimum width is 32. In version 9.4, SAS will automatically provide up to 3 decimal places for the seconds component of these values (indicated below by *fff*), regardless of significance (that is, if there is no decimal fraction of seconds, then zeros will be displayed). The number of decimal places shown is dependent upon the format width. The format will display the duration value as an ISO duration and an interval value as an ISO interval without any additional programming. SAS chooses one of the following forms based on the information stored in the ISO duration, datetime, or interval variable:

- P*n*Y*n*M*n*DT*n*H*n*M*n*S
- P*n*Y*n*M*n*DT*n*H*n*M*n*S/*yyyymmdd*T*hhmmssfff*
- *yyyymmdd*T*hhmmssfff*
- *yyyymmdd*T*hhmmssfff*/P*n*Y*n*M*n*DT*n*H*n*M*n*S
- *yyyymmdd*T*hhmmssfff*/*yyyymmdd*T*hhmmssfff*

$N8601BA*w.d*

$N8601BA*w*. writes duration, datetime, and interval values in the ISO 8601 basic notation. This format differs from the $N8601B. format in that it displays durations in the form P*yyyymmdd*T*hhmmss* instead of P*n*Y*n*M*n*DT*n*H*n*M*n*S.) It is left-justified, and *w* can be from 1 to 200, with a default width of 50. In order to display datetime and duration values correctly, the width must be at least 16, while to display interval values correctly, the minimum width is 32. If you attempt to display an interval with a width specification of less than 32, you will get a series of asterisks (*) as your output. In version 9.4, SAS will automatically provide up to 3 decimal places for the seconds component of these values (indicated below by *fff*), regardless of significance (that is, if there is no decimal fraction of seconds, then zeros will be displayed). Whether one, two, or three decimal places are shown is dependent upon the format width. The format will display the duration value as an ISO duration and an interval value as an ISO interval without any additional programming. SAS chooses one of the following forms based on the information stored in the ISO duration, datetime, or interval variable:

- P*yyyymmdd*T*hhmmss*
- *yyyymmdd*T*hhmmss*
- P*yyyymmdd*T*hhmmss*/*yyyymmdd*T*hhmmss*
- *yyyymmdd*T*hhmmss*/P*yyyymmdd*T*hhmmss*
- *yyyymmdd*T*hhmmss*/*yyyymmdd*T*hhmmss*

$N8601E*w*.

$N8601E*w*. writes duration, datetime, and interval values in the ISO 8601 extended notation. It is left-justified, and *w* can be from 1 to 200, with a default width of 50. In order to display datetime and duration values correctly, the width must be at least 16, while to display interval values correctly, the minimum width is 32. If you attempt to display an interval with a width specification of less than 32, you will get a series of asterisks (*) as your output. In version 9.4, SAS will automatically provide up to 3 decimal places for the seconds component of these values (indicated below by *fff*), regardless of significance (that is, if there is no decimal fraction of seconds, then zeros will be displayed). Whether one, two, or three decimal places are shown is dependent upon the format width. The format will display the duration value as an ISO duration and interval values as an ISO interval without any additional programming. SAS chooses one of the following forms based on the information stored in the ISO duration, datetime, or interval variable:

- P*n*Y*n*M*n*DT*n*H*n*M*n*S
- *yyyy-mm-dd*T*hh:mm:ss.fff*
- P*n*Y*n*M*n*DT*n*H*n*M*n*S/*yyyy-mm-dd*T*hh:mm:ss.fff*
- *yyyy-mm-dd*T*hh:mm:ss.fff*/P*n*Y*n*M*n*DT*n*H*n*M*n*S
- *yyyy-mm-dd*T*hh:mm:ss.fff*/*yyyy-mm-dd*T*hh:mm:ss.fff*

$N8601EA*w*.

$N8601EA*w*. writes duration, datetime, and interval values in the ISO 8601 extended notation. It is left-justified, and *w* can be from 1 to 200, with a default width of 50. In order to display datetime and duration values correctly, the width must be at least 16, while to display interval values correctly, the minimum width is 32. If you attempt to display an interval with a width specification of less than 32, you will get a series of asterisks (*) as your output. In version 9.4, SAS will automatically provide up to 3 decimal places for the seconds component of these values (indicated below by *fff*), regardless of significance (that is, if there is no decimal fraction of seconds, then zeros will be displayed). Whether one, two, or three decimal places are shown is dependent upon the format width. The format will display duration values as an ISO duration and interval values as an ISO interval without any additional programming. SAS chooses one of the following forms based on the information stored in the ISO duration, datetime, or interval variable:

- P*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*
- P*yyyy-mm-dd*T*hh:mm.ss.fff*/*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*/P*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*/*yyyy-mm-dd*T*hh:mm.ss.fff*

$N8601EH*w*.

$N8601EH*w*. writes duration, datetime, and interval values in the ISO 8601 extended notation, substituting hyphens for any missing components. With this format, omitted datetime components are always displayed. It is left-justified, and *w* can be from 1 to 200, with a default width of 50. In order to display datetime and duration values correctly, the width must be at least 16, while to display interval values correctly, the minimum width is 32. If you attempt to display an interval with a width specification of less than 32, you will get a series of asterisks (*) as your output. In version 9.4, SAS will automatically provide up to 3 decimal places for the seconds component of these values (indicated below by *fff*), regardless of significance (that is, if there is no decimal fraction of seconds, then zeros will be displayed). Whether one, two, or three decimal places are shown is dependent upon the format width. The format will display duration values as an ISO duration and interval values as an ISO interval without any additional programming, and it will display them in one of the following forms. SAS chooses one of the following forms based on the information stored in the ISO duration, datetime, or interval variable:

- P*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*
- P*yyyy-mm-dd*T*hh:mm.ss.fff*/*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*/P*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*/*yyyy-mm-dd*T*hh:mm.ss.fff*

$N8601EX*w*.

$N8601EX*w*. writes duration, datetime, and interval values in the ISO 8601 extended notation, substituting the letter "x" for any missing components. With this format, omitted datetime components are always displayed. It is left-justified, and *w* can be from 1 to 200, with a default width of 50. In order to display datetime and duration values correctly, the width must be at least 16, while to display interval values correctly, the minimum width is 32. If you attempt to display an interval with a width specification of less than 32, you will get a series of asterisks (*) as your output. In version 9.4, SAS will automatically provide up to 3 decimal places for the seconds component of these values (indicated below by *fff*), regardless of significance (that is, if there is no decimal fraction of seconds, then zeros will be displayed). Whether one, two, or three decimal places are shown is dependent upon the format width. The format will display duration values as an ISO duration and interval values as an ISO interval without any additional programming. SAS chooses one of the following forms based on the information stored in the ISO duration, datetime, or interval variable:

- P*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*
- P*yyyy-mm-dd*T*hh:mm.ss.fff*/*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*/P*yyyy-mm-dd*T*hh:mm.ss.fff*
- *yyyy-mm-dd*T*hh:mm.ss.fff*/*yyyy-mm-dd*T*hh:mm.ss.fff*

4.5.3 ISO 8601 Duration and Interval Informats

In order to store duration, interval, and datetime values that are already represented in their ISO 8601 form as character strings, you will need to use the following informats with the INPUT statement or INPUT() function. The CALL IS8601_CONVERT routine is designed to work with SAS date, time, and datetime values, not character strings.

$N8601B*w*.

$N8601B*w*. will translate duration, datetime, and interval strings written in the ISO 8601 basic or extended notation into the SAS internal representation for these values. Missing components are correctly processed as long as a single hyphen (-) is used in place of each missing component. *w* can be from 1 to 200, with a default width of 50. In order to process datetime and duration values correctly, *w* must be at least 16, while to process interval values correctly, *w* must be at least 32. The informat will process data in any one of the following forms:

ISO Form Read	Comment
P*yyyy-mm-dd*T*hh*:*mm*:*ss.fff*	This is a duration value, not a datetime. Therefore, it does not indicate a specific datetime but an event that extends over the time indicated.
P*yyyymmdd*T*hhmmss*	This is a duration value, not a datetime. Therefore, it does not indicate a specific datetime but an event that extends over the time indicated.
P*n*Y*n*M*n*DT*n*H*n*M*n.fff*S	
P*n*W	
*yyyy-mm-dd*T*hh*:*mm*:*ss.fff*/*yyyy-mm-dd*T*hh*:*mm*:*ss.fff*	
*yyyymmdd*T*hhmmssfff*/*yyyymmdd*T*hhmmssfff*	
P*n*Y*n*M*n*DT*n*H*n*M*n.fff*S/*yyyy-mm-dd*T*hh*:*mm*:*ss.fff*	
*yyyy-mm-dd*T*hh*:*mm*:*ss.fff*/P*n*Y*n*M*n*DT*n*H*n*M*n.fff*S	
*yyyy-mm-dd*T*hh*:*mm*:*ss.fff*	
*yyyymmdd*T*hhmmss.fff*	

$N8601E*w*.

$N8601E*w*. will translate duration, datetime, and interval strings written in the ISO 8601 extended notation into the SAS internal representation for these values. This informat differs from the $N8601B. informat in that it will only process values that are in the ISO 8601 extended format. If you attempt to read a value in the basic notation with this informat, you will get an error, and the stored result will be a missing value. This informat is useful when you need to enforce adherence to the extended notation. Missing components are correctly processed as long as a single hyphen (-) is used in place of each missing component. *w* can be from 1 to 200, with a default width of 50. In order to process datetime and duration values correctly, *w* must be at least 16, while to process interval values correctly, *w* must be at least 32.

To demonstrate the difference between the $N8601B. and $N8601E. informats, the example below will use the same duration string in both the basic and extended notations and try to process it with each informat.

Example 4.2: The $N8601B. Informat versus the $N8601E. Informat

```
stored = INPUTC(duration_string,inf);
```

Example 4.2 uses the above code to store the duration string. *inf* represents the informat being used.

Reading the ISO 8601 Basic Notation

Informat Used	Duration String	Stored Representation	Formatted Representation
$N8601B.	P00020806T0100	00028060100FFFFC	P2Y8M6DT1H0M
$N8601E.	P00020806T0100		*****************

As you can see, the $N8601E. informat failed to process the basic notation string, resulting in a missing value. When you try to process a string in the ISO basic notation using the extended notation-specific informat, you will get the following note in your log.

```
NOTE: Invalid argument to function INPUT at line xxx column yy.
```

Now let's see what happens when we try to process the extended notation string.

Reading the ISO 8601 Extended Notation

Informat Used	Duration String	Stored Representation	Formatted Representation
$N8601B.	P0002-08-06T01:00	00028060100FFFFC	P2Y8M6DT1H0M
$N8601E.	P0002-08-06T01:00	00028060100FFFFC	P2Y8M6DT1H0M

In this case, both informats caused SAS to store identical values. The $N8601B. informat can process both basic and extended notations, but the $N8601E. informat will only work with the extended notation. Use the $N8601E. informat if you want to make sure that the input data conform to the extended notation.

4.5.4 CALL IS8601_CONVERT

It is important to understand that although ISO durations and intervals are stored in character variables, they are not simple text. You cannot convert a datetime to a character string using the PUT() function and then concatenate it with a duration string or another datetime that has been converted to a character string. In order to use the SAS date and time capacity with ISO durations and intervals, you must first convert ISO duration and interval values to an internal representation that will allow this.

The formats and informats we just discussed will work on character values, but what if you have SAS date or datetime values, which of course are stored in numeric variables, and you want to use those to build your ISO durations and intervals? The CALL IS8601_CONVERT routine performs this conversion in both directions, so it will create your ISO durations and intervals from those SAS

date and datetime values. It will also do the reverse, and transform your ISO durations and intervals into their individual date and datetime components. In order to create an interval or a duration, you will need two values: a start and an end (defined as dates or datetimes), or you can provide a duration, and either start or an end. Similarly, in order to convert an interval, you will be splitting the components into two variables, a duration and a datetime, or two datetimes. In addition, the routine can derive the starting or ending point of an ISO duration or interval.

The syntax for the CALL IS8601_CONVERT routine is:

CALL IS8601_CONVERT(convert-from,convert-to,from--variables,to-variables, replacements);

The *convert-from* argument describes the type of date/datetime/interval/duration data that you are converting. *convert-from* can be one of the following keyword values, enclosed in quotation marks, or might also be represented by a character variable that resolves to one of these values.

Keyword Value	**Description**
'd*n*'	Use this when the value that you want to convert consists of individual components of a date value. *n* is from 1 to 6, and indicates how many components are in the list of *from-variables*. You can create either a date or a datetime value with this argument type. The individual components from left to right are: month, day, year, hour, minute, and second.
'dt*n*'	Use this when the value that you want to convert are individual components of a datetime value. *n* is from 1 to 6, and indicates how many components are in the list of *from-variables*. You can create either a date or a datetime value with this argument type. The individual components from left to right are: month, day, year, hour, minute, and second.
'du*n*'	Use this when you want to convert a value consisting of individual components of a duration value. *n* is from 1 to 6, and indicates how many components are in the duration value. The individual components from left to right are the number of months, days, years, hours, minutes, and seconds in the duration.
'dt/dt'	Use this when you are converting two datetime values.
'dt/du'	Use this when you are converting a datetime/duration interval. This signifies that the datetime is the start of the interval that you are converting.
'du/dt'	Use this to convert a duration/datetime interval. This signifies that the datetime is the end of the interval that you are converting.
'intvl'	Use this to convert interval values.

The *convert-to* argument describes the form of the result from the CALL IS8601_CONVERT routine.

Keyword Value	**Description**
'intvl'	Use this to create an interval value.
'dt/dt'	Use this to create a datetime/datetime interval.
'dt/du'	Use this to create a datetime/duration interval. This signifies that the datetime is the beginning of the interval.
'du/dt'	Use this to create a duration/datetime interval. This signifies that the datetime is the end of the interval.
'du'	Use this to create a duration.
'start'	Use this to derive a starting date or duration from an interval value.
'end'	Use this to derive the ending date or duration from an interval value.
'd*n*'	Use this when you want to create individual components of a date value. *n* is from 1 to 6, and indicates how many components are in the list of *from-variables*. The individual components will be stored from left to right in the following order: month, day, year, hour, minute, and second.
'dt*n*'	Use this when you want to create individual components of a datetime value. *n* is from 1 to 6, and indicates how many components are in the list of *from-variables*. The individual components will be stored from left to right in the following order: month, day, year, hour, minute, and second.
'du*n*'	Use this to create individual components of a duration value. *n* is from 1 to 6, and indicates how many components are in the list of *from-variables*. The individual components will be stored from left to right in this order: the number of months, days, years, hours, minutes, and seconds.

from-variables are the variables containing the value(s) to be converted. Specify one variable if you are converting an interval and two for datetime and/or duration values. If you are converting individual components of a date or a datetime (using 'd*n*,' 'dt*n*,' or 'du*n*' in your convert-from argument), then you will need one variable for each component specified. If you are converting an interval, the variable must be at least 32 characters in length.

to-variables are the variables containing the result that the routine calculates. Specify one variable if you are creating an interval and two for datetime and/or duration values. If you want individual components of a date or a datetime from the function (using 'd*n*,' 'dt*n*,' or 'du*n*' in your convert-to argument), then you will need one variable for each component specified.

replacements enables you to provide your own value for the month, day, hour, minute, and second components to be used if any of those components are missing in the *convert-from* argument. The default is 1 for month and day, and 0 for hour, minute, and second. When you use this parameter, even though the year component is required for a valid ISO duration or interval value, you will need a leading comma as a placeholder for the year value, followed by all 5 replacement values, separated by commas. While it is possible to leave some of the replacement values blank (and

thereby use the default), it is easier to read when a value is provided for each of the replacements, even if it is the same as the default.

The following examples demonstrate many of the capabilities of the CALL IS8601_CONVERT routine, including the ability to perform date and time calculations.

Example 4.3: How Long Is… in SAS time?

Most of the calculations involving SAS dates and times have a specific reference point in mind; in general, there is a date and/or a time involved, such as June 8, 2014, at 3:50 p.m. However, what can you do if you want to measure an ISO duration in SAS time? One possibility is to use a sample starting point and SAS intervals to obtain the endpoint. From there, it is just a matter of subtracting your dummy start point from your endpoint. Or you could use CALL IS8601_CONVERT. The first period is simple: How long is four weeks expressed in time? The second period asks the same question for 3 days, 4 hours, 27 minutes, and 16.8 seconds, which is not so easy to do with SAS intervals.

```
1.  DATA howlong;
2.  LENGTH period $ 16;
3.  period = "P4W";
4.  CALL IS8601_CONVERT('du','du',period,howlong);
5.  howlong_disp = STRIP(PUT(howlong,time10.1));
6.  OUTPUT;
7.  period = "P3DT4H27M16.8S";
8.  CALL IS8601_CONVERT('du','du',period,howlong);
9.  howlong_disp = STRIP(PUT(howlong,time10.1));
10. OUTPUT;
11. RUN;
```

In lines 4 and 8, we use 'du' as the *convert-to* argument because there is no argument value that will explicitly give a time value. However, when you use the 'du' argument and do not specify the result as a character variable, CALL IS8601_CONVERT will create your duration in seconds (that is, a SAS time value). Here is the PROC PRINT of our HOWLONG data set:

ISO 8601 Duration	Time in Seconds	Time in Seconds Formatted as TIME10.1
P4W	2419200.0	672:00:00.0
P3DT4H27M16.8S	275236.8	76:27:16.8

Example 4.4: Converting Two Datetimes to an ISO Duration

This situation is one frequently encountered in clinical trials. The starting and ending datetimes for an event would be presented as text, and you need to calculate the duration and display it in ISO 8601 format according to the CDISC specification. In this example, we will take simulated event data (some of which has missing components and entire missing values), calculate the durations, and present the result according to the ISO standard. Here are the data shown in their original text representation.

Obs	Starting Date	Ending Date
1		
2	2013-01-18T09:30	2013-01-18T21:00
3	2012-12-30T11	2013-01-01T14:00
4	2012-11	2013-01-04
5	2012-11-19	2013-01-04
6	2012-11-20T08:30	2013-01-03
7	2012	2013-02
8	2012-12-27	
9	2012-11	2012-11-20
10	2012-11-19T14:15	2012-11-20
11	2012-12-17T08:20	2013-01-02T09

The first step is to create these as SAS datetime values. Let's run a quick SAS program to create our example data set, AEDTM:

```
DATA book.aedtm;
INFILE "eventdata.txt" PAD MISSOVER DLM='09'x FIRSTOBS=2 DSD;
INPUT aestdtm :E8601DT. aeendtm :E8601DT.;
FORMAT aestdtm aeendtm datetime20.;
RUN;
```

Executing this code results in this data set:

VIEWTABLE: Book.Aedtm   Print

	aestdtm	aeendtm
1	.	.
2	18JAN2013:09:30:00	18JAN2013:21:00:00
3	.	01JAN2013:14:00:00
4	.	.
5	.	.
6	20NOV2012:08:30:00	.
7	.	.
8	.	.
9	.	.
10	19NOV2012:14:15:00	.
11	17DEC2012:08:20:00	.
12	.	.

What happened here? Most of the datetime values are missing! This is a direct consequence of incomplete data for dates and times in the source data. Remember that SAS datetime values are a complete date and a complete time, so yes, the values are missing because there is no complete date and time in most of these cases. The ISO standard can accommodate partial dates and times, but the normal SAS date and time facility cannot. Let's try this a different way using the ISO8601 informats.

```
DATA book.aedtm2;
INFILE "eventdata.txt" PAD MISSOVER DLM='09'x FIRSTOBS=2 DSD;
INPUT aestdtm :$N8601B. aeendtm :$N8601B.;
FORMAT aestdtm aeendtm datetime20.;
RUN;
```

This code didn't even run, giving an error in the log.

```
1   DATA book.aedtm2;
2   INFILE "eventdata.txt" PAD MISSOVER DLM='09'x FIRSTOBS=2 DSD;
3   INPUT aestdtm :$N8601B. aeendtm :$N8601B.;
4   FORMAT aestdtm aeendtm datetime20.;
                           -----------
                             48
ERROR 48-59: The format $DATETIME was not found or could not be
loaded.

5   RUN;

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set BOOK.AEDTM2 may be incomplete.  When this step
         was stopped there were 0 observations and 2 variables.
```

The ISO informats only create character variables. Therefore, the error arises when you try to use a numeric format with a character variable. At this point, we are no closer to getting our durations than before. What happens when we take the FORMAT statement out?

VIEWTABLE: Book.Aedtm2

	aestdtm	aeendtm
1	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
2	20131180930FFFFD	20131182100FFFFD
3	2012C3011FFFFFFD	20131011400FFFFD
4	2012BFFFFFFFFFFD	2013104FFFFFFFFD
5	2012B19FFFFFFFFD	2013104FFFFFFFFD
6	2012B200830FFFFD	2013103FFFFFFFFD
7	2012FFFFFFFFFFFD	20132FFFFFFFFFFD
8	2012C27FFFFFFFFD	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
9	2012BFFFFFFFFFFD	2012B20FFFFFFFFD
10	2012B191415FFFFD	2012B20FFFFFFFFD
11	2012C170820FFFFD	201310209FFFFFFD

Now we've got the internal representation of ISO datetimes in a SAS data set, which we have named AEDTM2. Let's format the two variables using the $N8601EH. format so that we can see the result with a proper ISO representation. The $N8601EH. format provides hyphens for missing components.

VIEWTABLE: Book.Aedtm2

	aestdtm	aeendtm
1		
2	2013-01-18T09:30:-	2013-01-18T21:00:-
3	2012-12-30T11:-:-	2013-01-01T14:00:-
4	2012-11–T-:-:-	2013-01-04T-:-:-
5	2012-11-19T-:-:-	2013-01-04T-:-:-
6	2012-11-20T08:30:-	2013-01-03T-:-:-
7	2012—T-:-:-	2013-02–T-:-:-
8	2012-12-27T-:-:-	
9	2012-11–T-:-:-	2012-11-20T-:-:-
10	2012-11-19T14:15:-	2012-11-20T-:-:-
11	2012-12-17T08:20:-	2013-01-02T09:-:-

From here, we can calculate durations with CALL IS8601_CONVERT using the following code and the data set AEDTM2 that we created in the previous step. In line 3, we define the result variables as character so that we do not get a SAS time value. We are creating two for the purposes of this exercise, DURATION with the routine in line 4, and AEDUR in line 6, which is the ISO-formatted version of DURATION so that it can be shown alongside the internal representation of the duration.

```
1.  DATA book.iso_durations;
2.  SET book.aedtm2;
3.  LENGTH duration aedur $ 16;
4.  CALL IS8601_CONVERT('dt/dt','du',aestdtm,aeendtm,duration);
```

```
5.  aedur = duration;
6.  FORMAT aedur $N8601E.;
7.  RUN;
```

VIEWTABLE: Book.Iso_durations

	aestdtm	aeendtm	duration	aedur
1				
2	2013-01-18T09:30:-	2013-01-18T21:00:-	FFFFFFF1130FFFFC	PT11H30M
3	2012-12-30T11:-:-	2013-01-01T14:00:-	FFFFF0203FFFFFFC	P2DT3H
4	2012-11--T-:-:-	2013-01-04T-:-:-	FFFF204FFFFFFFFC	P2M4D
5	2012-11-19T-:-:-	2013-01-04T-:-:-	FFFF115FFFFFFFFC	P1M15D
6	2012-11-20T08:30:-	2013-01-03T-:-:-	FFFF1121530FFFFC	P1M12DT15H30M
7	2012---T-:-:-	2013-02--T-:-:-	00012FFFFFFFFFFC	P1Y2M
8	2012-12-27T-:-:-			
9	2012-11--T-:-:-	2012-11-20T-:-:-	FFFFF20FFFFFFFFC	P20D
10	2012-11-19T14:15:-	2012-11-20T-:-:-	FFFFFFF0945FFFFC	PT9H45M
11	2012-12-17T08:20:-	2013-01-02T09:-:-	FFFFF16FF40FFFFC	P16DT40M

One of the things that you might notice is that durations have been calculated for all the values that have an ending date, regardless of missing components in the starting and ending datetimes. The CALL IS8601_CONVERT routine imputes complete datetime values by using default values for missing components, so it has supplied 1 for missing months and days, and zeros for hours, minutes, and seconds. This causes any datetime with a missing time to be set at 12:00 a.m. of the following day. This simple imputation might not be what you want to use, and more elaborate coding might be necessary. You might have to start with parsing the input datetime string to determine which components are missing and apply your imputation algorithm from there. Nonetheless, once you have your imputed datetime values, CALL IS8601_CONVERT will create and store an ISO duration from those datetimes.

Example 4.5: Converting Two Datetimes to an ISO Interval

Let's use the data from example 4.4 to quickly demonstrate the creation of an ISO interval using the CALL IS8601_CONVERT routine. We will start with the ISO datetime values stored in the data set AEDTM2.

VIEWTABLE: Book.Aedtm2

	aestdtm	aeendtm
1		
2	2013-01-18T09:30:-	2013-01-18T21:00:-
3	2012-12-30T11:-:-	2013-01-01T14:00:-
4	2012-11--T-:-:-	2013-01-04T-:-:-
5	2012-11-19T-:-:-	2013-01-04T-:-:-
6	2012-11-20T08:30:-	2013-01-03T-:-:-
7	2012---T-:-:-	2013-02--T-:-:-
8	2012-12-27T-:-:-	
9	2012-11--T-:-:-	2012-11-20T-:-:-
10	2012-11-19T14:15:-	2012-11-20T-:-:-
11	2012-12-17T08:20:-	2013-01-02T09:-:-

The following code will create our ISO interval. Note that in line 3, the length of the result variable interval is set to 32, which is the minimum length for a variable that holds an ISO interval. The result is shown in the screen capture following the code. Once again, we are creating a copy of the result to show both the interval value as a stored value and the ISO-formatted interval value.

```
1.  DATA book.iso_intervals;
2.  SET book.aedtm2;
3.  LENGTH interval formatted_interval $ 32;
4.  CALL IS8601_CONVERT('dt/dt','intvl',aestdtm,aeendtm,interval);
5.  formatted_interval = interval;
6.  FORMAT formatted_interval $N8601E.;
7.  RUN;
```

VIEWTABLE: Book.Iso_intervals

	aestdtm	aeendtm	interval	formatted_interval
1				
2	2013-01-18T09:30:-	2013-01-18T21:00:-	20131180930FFFFD20131182100FFFF	2013-01-18T09:30/2013-01-18T21:00
3	2012-12-30T11:-:-	2013-01-01T14:00:-	2012C3011FFFFFFD20131011400FFFF	2012-12-30T11/2013-01-01T14:00
4	2012-11–T-:-:-	2013-01-04T-:-:-	2012BFFFFFFFFFFFD2013104FFFFFFF	2012-11/2013-01-04
5	2012-11-19T-:-:-	2013-01-04T-:-:-	2012B19FFFFFFFFD2013104FFFFFFF	2012-11-19/2013-01-04
6	2012-11-20T08:30:-	2013-01-03T-:-:-	2012B200830FFFFD2013103FFFFFFFF	2012-11-20T08:30/2013-01-03
7	2012—T-:-:-	2013-02–T-:-:-	2012FFFFFFFFFFFFD20132FFFFFFFFF	2012/2013-02
8	2012-12-27T-:-:-			
9	2012-11–T-:-:-	2012-11-20T-:-:-	2012BFFFFFFFFFFFD2012B20FFFFFFF	2012-11/2012-11-20
10	2012-11-19T14:15:-	2012-11-20T-:-:-	2012B191415FFFFD2012B20FFFFFFFF	2012-11-19T14:15/2012-11-20
11	2012-12-17T08:20:-	2013-01-02T09:-:-	2012C170820FFFFD201310209FFFFFF	2012-12-17T08:20/2013-01-02T09

Example 4.6: Converting a Datetime and a Duration into an ISO Interval

In this example, we will create an ISO interval from a datetime and a duration using the following data set, SAMPLE, which contains a start datetime and an ISO duration.

VIEWTABLE: Work.Sample

	aestdtm	duration
1	2012B221030FFFFD	FFFFF072330FFFFC
2	2012C02FFFFFFFFD	FFFF105FFFFFFFFC
3	2012C121200FFFFD	FEFFFFF12FFFFFFC
4	2012C191000FFFFD	FFFFF072130FFFFC
5	2013101FFFFFFFFD	FFFFF29FFFFFFFFC

Why don't we apply the $N8601E. format to AESTDTM and DURATION so that they are readable?

VIEWTABLE: Work.Sample

	aestdtm	duration
1	2012-11-22T10:30	P7DT23H30M
2	2012-12-02	P1M5D
3	2012-12-12T12:00	-PT12H
4	2012-12-19T10:00	P7DT21H30M
5	2013-01-01	P29D

If this were true clinical data, there would be an error in the data at record 3, because the duration is negative, indicating that the start datetime is later than the ending datetime. While the ISO standard accounts for negative durations, this does not mean that negative durations are appropriate in your data.

This code looks similar to the code in the previous two examples. The difference is in the arguments used for the CALL IS8601_CONVERT routine in line 4. The 'dt/du' signifies that the datetime represents the start of the interval. We will show the unformatted and formatted result as a reminder that the character results are stored in their SAS internal representation:

```
1.  DATA interval;
2.  SET sample;
3.  LENGTH interval formatted_interval $ 32;
4.  CALL IS8601_CONVERT('dt/du','intvl',aestdtm,duration,interval);
5.  formatted_interval = interval;
6.  FORMAT formatted_interval $N8601E.;
7.  RUN;
```

Interval	Formatted_Interval
2012B221030FFFFDFFFFF072330FFFFC	2012-11-22T10:30/P7DT23H30M
2012C02FFFFFFFFDFFFF105FFFFFFFFC	2012-12-02/P1M5D
2012C121200FFFFDFEFFFFF12FFFFFFC	2012-12-12T12:00/-PT12H
2012C191000FFFFDFFFFF072130FFFFC	2012-12-19T10:00/P7DT21H30M
2013101FFFFFFFFDFFFFF29FFFFFFFFC	2013-01-01/P29D

Example 4.7: Calculating the End of an Interval from a Datetime and a Duration

An additional capability of the CALL ISO_8601 routine is to calculate starting and ending datetimes from intervals, or a combination of duration and datetime. In this example, we will calculate the end datetimes for the ISO intervals that we created in example 4.6.

```
1.  DATA interval_end;
2.  SET interval;
3.  CALL IS8601_CONVERT('intvl','end',interval,end);
4.  FORMAT interval $N8601E. end DATETIME20.;
5.  RUN;
```

You can see that this code is again slightly different from example 4.6. One large difference is that there is no LENGTH statement used to create the result as a character variable. This causes the routine to create the result as a SAS datetime value, not an ISO 8601 datetime character string. This is an important distinction. In line 3, we tell the routine that we are going to convert an interval, and we want the ending date based on that interval. Since the variable INTERVAL is still stored in its internal SAS representation, we have applied the $N8601E. format to that variable, as well as formatting the END variable with DATETIME20. so we can read them easily in the output. Here is the result.

Interval	End
2012-11-22T10:30/P7DT23H30M	30NOV2012:10:00:00
2012-12-02/P1M5D	07JAN2013:00:00:00
2012-12-12T12:00/-PT12H	11DEC2012:12:00:00
2012-12-19T10:00/P7DT21H30M	27DEC2012:07:30:00
2013-01-01/P29D	30JAN2013:00:00:00

Any of the preceding examples can be reversed; you can convert an ISO duration or interval into two datetimes or you can turn an interval into a duration and a datetime. To show you how it all connects to together, in this last example, we will take starting and ending ISO datetime values and create an ISO duration and an ISO interval, and then we will recalculate the starting and ending datetimes from that ISO interval, both as their ISO datetime values and their SAS datetime values.

Example 4.8: CALL IS8601_CONVERT: ISO Datetimes, Durations, and Intervals from Start to End and Back Again with One Routine

This code uses the data set EX4_8, which contains two ISO datetime values in the variables AESTDTM and AEENDTM. They are displayed using the $N8601E. format so you can read the values easily.

VIEWTABLE: Work.Ex4_8

	aestdtm	aeendtm
1	2012-11-21T09:12	2012-11-30
2	2012-11-21T09:25	2012-11
3	2012-12-03T09:00	2012-12-03T10:00
4	2012-12-14T09:12	2013-01-17
5	2012-12-14	2012-12-29
6	2012-12-14	
7	2012-12-15	2013-01-11
8	2012-12-19T08:00	2013-01-02T09:00
9	2012-12-26	2013-01-15
10	2013-01-12T17:00	2013-01-18T09:00

Because we want the results in this example as their ISO representations, we define the result variables that we are creating as character variables, including the ending datetimes. Note that the variable INTERVAL is 32 bytes in length because that is the minimum length for an ISO interval value. Line 4 creates the ISO duration and line 5 the ISO interval value, and we use that interval in lines 6 and 7 to produce the starting and ending ISO datetime values. Lines 8 and 9 lines recalculate the starting and ending datetimes as SAS datetime values. We will present the results separately.

```
1.  DATA circle;
2.  SET ex4_8;
3.  LENGTH duration $ 16 interval $ 32 start_dt $ 16 end_dt $ 16;
4.  CALL IS8601_CONVERT('dt/dt','du',aestdtm,aeendtm,duration);
5.  CALL IS8601_CONVERT('dt/dt','intvl',aestdtm,aeendtm,interval);
6.  CALL IS8601_CONVERT('intvl','start',interval,start_dt);
7.  CALL IS8601_CONVERT('intvl','end',interval,end_dt);
8.  CALL IS8601_CONVERT('intvl','start',interval,datetime_start);
9.  CALL IS8601_CONVERT('intvl','end',interval,datetime_end);
10. FORMAT duration interval start_dt end_dt $N8601E.;
11. RUN;
```

Original Starting ISO Datetime	Original Ending ISO Datetime	ISO Duration Value	ISO Interval Value
2012-11-21T09:12	2012-11-30	P8DT14H48M	2012-11-21T09:12/2012-11-30
2012-11-21T09:25	2012-11	***-P21DT9H25M***	2012-11-21T09:25/2012-11

Original Starting ISO Datetime	Original Ending ISO Datetime	ISO Duration Value	ISO Interval Value
2012-12-03T09:00	2012-12-03T10:00	PT1H	2012-12-03T09:00/2012-12-03T10:00
2012-12-14T09:12	2013-01-17	P1M2DT14H48M	2012-12-14T09:12/2013-01-17
2012-12-14	2012-12-29	P15D	2012-12-14/2012-12-29

The one thing that you should take note of here is the negative duration value in the second row. Since the day component is missing from the ending datetime, CALL IS8601_CONVERT has used its default value for the day component, which is 1. Obviously, November 21 (the starting date) is after November 1, so you might want to use a different imputation for your ending dates. Next are the recalculated starting and ending dates, presented in separate tables to make it easy to see the result of the conversions.

Original Starting ISO Datetime	Recalculated ISO Starting Datetime	Recalculated SAS Starting Datetime
2012-11-21T09:12	2012-11-21T09:12	21NOV2012:09:12:00
2012-11-21T09:25	2012-11-21T09:25	21NOV2012:09:25:00
2012-12-03T09:00	2012-12-03T09:00	03DEC2012:09:00:00
2012-12-14T09:12	2012-12-14T09:12	14DEC2012:09:12:00
2012-12-14	2012-12-14	14DEC2012:00:00:00

Original Ending ISO Datetime	Recalculated ISO Ending Datetime	Recalculated SAS Ending Datetime
2012-11-30	2012-11-30	30NOV2012:00:00:00
2012-11	*2012-11*	*01NOV2012:00:00:00*
2012-12-03T10:00	2012-12-03T10:00	03DEC2012:10:00:00
2013-01-17	2013-01-17	17JAN2013:00:00:00
2012-12-29	2012-12-29	29DEC2012:00:00:00

Take note of the second row in the ending datetime table above. Not only does this point out the problem with relying on the automatic substitution of the routine, but the difference between the ISO datetime values and SAS datetime values. The ISO standard accommodates missing components, so it is legal to have the value "2012-11," which clearly indicates, "During the month of November, but the exact day is unknown," and, "At some time on the day of…" The SAS

datetime value has to provide both exact date and exact time, so "2012-11" is set to 12:00 AM, November 1, 2012, using the default replacements for missing components.

4.6 Conclusion

The ISO 8601 standard has been developed for the clear communication of date and time information across countries, applications, and platforms. SAS has the capability to handle this standard to its full extent, even though on the surface there seem to be many incompatibilities between the SAS date and time facility and the ISO standard. The largest differences are with respect to incomplete data, and that alphabetic characters are used as more than simple delimiters makes the ISO 8601 handling facility in SAS unique when it comes to dates and times. It is the only piece of the SAS date and time capabilities that requires character variables.

Chapter 5: Date and Time Functions

5.1 Current Date and Time Functions 137
5.2 Extracting Pieces from SAS Date, Time, and Datetime Values 138
5.3 Creating Dates, Times, and Datetimes from Numbers or Other Information 140
5.4 Calculating Elapsed Time, and the HOLIDAY() Function 145
5.5 The Basics of SAS Intervals 149
5.6 Modifying SAS Intervals 159
5.7 Creating Your Own SAS Intervals 169
5.8 Interval Functions about Intervals... 176
5.9 Retail Calendar Intervals and Seasonality 181

SAS has many functions to manipulate dates, times, and datetime values. The functions can be categorized according to what they do. You can obtain the current date, time, or datetime (as specified by the computer's clock). You can also easily extract pieces of dates, times, or datetimes as numerical values from their corresponding SAS values, or you can assemble SAS date, time, and datetime values from SAS variables or constants. Another set of functions operates with intervals such as weeks or months.

5.1 Current Date and Time Functions

Current date and time functions have no arguments and return SAS values as noted in the following table. The values are obtained from the operating system's clock.

Table 5.1: Current Date and Time Functions

Date and Time Function	Description
DATE( ), TODAY( )	These functions are identical, and both return the current date as a SAS date value.

Date and Time Function	Description
TIME()	This returns the current time as a SAS time value.
DATETIME()	This returns the current date and time as a SAS datetime value.

5.2 Extracting Pieces from SAS Date, Time, and Datetime Values

The extraction functions all use a single argument (represented by *arg* in the following table), which represents a SAS date, time, or datetime value. This can be either a SAS variable name or the appropriate constant. If two-digit year values are used, the result will be subject to the YEARCUTOFF= option value in effect. They all return a numeric value as the result. The following tables are separated into functions that use dates as an argument, those that use datetimes as the argument, and those that use times as an argument. Each table gives examples of how to apply each function, along with relevant comments for each example.

Table 5.2: Functions Returning a Date Component and Requiring a SAS Date Value as an Argument

Function Name	Explanation	Example
DAY*(arg)*	Extracts the number of the day of the month from a SAS date value.	DAY("14OCT2015"d) = 14
JULDATE*(arg)*	Extracts the Julian date from a SAS date value. It will return a four- or five-digit value with a one- or two-digit year, if the year portion of the date falls within the 100-year span defined by the YEARCUTOFF= option. If you want to ensure four-digit year values, you should use the JULDATE7() function.	JULDATE("09MAY2004"d) = 4130 (result is returned as a numeric value, so there are no leading zeros) JULDATE("09MAY2014"d)=14129 (note difference caused by 2014 versus 2004, above) JULDATE("09MAY1890"d) = 1890129
JULDATE7*(arg)*	Extracts the Julian date with a four-digit year from a SAS date value. This always returns a seven-digit number, regardless of the year.	JULDATE7("09MAY2004"d) = 2004130 JULDATE7("09MAY2014"d) = 2014129 JULDATE7("09MAY1890"d) = 1890129
MONTH*(arg)*	Extracts the numerical month from a SAS date value.	MONTH("22AUG2015"d) = 8
QTR*(arg)*	Extracts the quarter of the year from a SAS date value.	QTR("8JAN2013"d) = 1

Function Name	Explanation	Example
WEEK(*arg*)	Extracts the week number from a SAS date value, where Sunday is the first day of the week, which is the "U" algorithm. This function has been augmented in SAS versions 9.1.3 and above by the WEEK(*arg*,descriptor) function described immediately below.	WEEK("02JAN2005"d) = 1
(Version 9.1.3 and up) WEEK(*arg*,*descriptor*)	Extracts the week number from a SAS date value. *descriptor* can be "U," "V," or "W" (case-insensitive), and it refers to the algorithm used to calculate the first week of the year.	
	The U algorithm calculates weeks based on Sunday being the first day of the week.	WEEK("02JAN2005"d,"U") = 1. January 2, 2005 was a Sunday, so the first week of the year has started.
	The V algorithm calculates weeks to the ISO standard. Monday is the first day of the week, and the first week of the year is defined as the one that contains both January 4 and the first Thursday of the year.	WEEK("02JAN2005"d,"V") = 53. This week is defined as being the 53rd week in 2004, because it doesn't contain the first Monday or Thursday of the year.
	The W algorithm calculates weeks based on Monday being the first day of the week without restriction.	WEEK("02JAN2005"d,"W ") = 0. The year 2005 has started, but weeks are calculated with Monday as the first day of the week. Therefore, the first week of 2005 doesn't start until January 3, 2005, so this is week 0 of 2005.
WEEKDAY(*arg*)	Extracts the number of the day of the week, where Sunday=1, Monday=2, and so on from a SAS date value.	WEEKDAY("14APR2011" d) = 4 (Wednesday, April 14, 2011)
YEAR(*arg*)	Extracts the year from a SAS date value. If you use a date constant (as in the example) and not a SAS date value, it is important to remember that the	If OPTIONS YEARCUTOFF=1920; YEAR("19JUL10"d) = 2010; YEAR("19JUL1910"d) = 1910

Function Name	Explanation	Example
	YEARCUTOFF= option affects two-digit years.	

Table 5.3: Functions Returning a Datetime Component and Requiring a SAS Datetime Value as an Argument

Function Name	Explanation	Example
DATEPART*(arg)*	Extracts the date from a SAS datetime value as a SAS date value.	DATEPART('21MAR2012:17:07:00'dt) = 19073 (March 21, 2012)
TIMEPART*(arg)*	Extracts the time portion from a SAS datetime value as a SAS time value.	TIMEPART("06SEP2012:13:36:33"dt) = 48993 (1:36:33 PM)

Table 5.4: Functions Returning a Time Component and Requiring a SAS Time Value as an Argument

Function Name	Explanation	Example
HOUR(*arg*)	Extracts the hour from a SAS time value.	HOUR("7:35:00"t) =7
MINUTE(*arg*)	Extracts the minutes from a SAS time value.	MINUTE("12:17:43 PM"t) = 17
SECOND(*arg*)	Extracts the seconds from a SAS time value.	SECOND("2:17:43"t) = 43

5.3 Creating Dates, Times, and Datetimes from Numbers or Other Information

5.3.1 Introduction

This series of functions will create SAS date, time, and datetime values from numerical variables or constants. While informats take complete date, time, and datetime references and translate them to their corresponding SAS value, these functions will create a SAS value from discrete pieces such as month, day, and year.

5.3.2 List of Functions and Their Descriptions

DATEJUL(Julian-date);

DATEJUL(Julian-date); creates a SAS date value from a numeric value representing a Julian date. *Julian-date* must be of the type *yy(yy)ddd*, where *yy(yy)* is two or four digits representing the year, and *ddd* must be a number from 1 to 365 (366 if a leap year). If you use two digits for the year, the YEARCUTOFF= option will be used to determine the century. The following table gives examples of how to apply this function:

Sample Function Call	SAS Date Value	Formatted with MMDDYY10. Format	Comments
OPTIONS YEARCUTOFF=1920; DATEJUL(21286)	–13959	10/13/1921	With the YEARCUTOFF value of 1920, the 21 is interpreted as 1921.
OPTIONS YEARCUTOFF=2000; DATEJUL(21286)	22566	10/13/2021	If YEARCUTOFF is 2000, the 21 is interpreted as 2021.
DATEJUL(2014174)	19897	06/23/2014	
DATEJUL(1989005)	10597	01/05/1989	
DATEJUL(00368)	.	.	368 is not a valid value for a Julian day, so the function returns a missing value.

DHMS(date,hour,minute,second);

DHMS(date,hour,minute,second); creates a SAS datetime value. All four arguments are required. *date* is a SAS date value, which can be either a numeric value or a date constant. If you use a two-digit year in a date constant, the date will be translated according to the YEARCUTOFF= option. *hour*, *minute*, and *second* are all numeric variables and/or constants. *Hour*, *minute*, and *second* are not restricted to their clock times. Therefore, *hour* can be greater than 24, while *minute* and *second* can be greater than 60. The following table gives examples of how to apply this function:

Sample Function Call	Datetime Value	Formatted with DATETIME19. Format	Comments
DHMS("08JUN2015"d,15,24,0)	1749396240	08JUN2015:15:24:00	
DHMS("02FEB2014"d,11,54,15)	1706961255	02FEB2014:11:54:15	
DHMS("30JUN2012"d,8,7,93)	1656662913	30JUN1992:08:08:33	The value 93 is just an argument. The function ultimately returns the datetime value in seconds and the formatted value converts the result. Therefore, 93 seconds becomes 1 minute, 33 seconds, which adds 1 to the minute value of 7, and reduces the seconds to 33.

Example 5.1: Using DHMS() When You Already Have a SAS Date and Time

If you need to create a datetime from a SAS date and a SAS time value, you do not have to use the HOUR(), MINUTE(), and SECOND() functions to extract those components from the time value. Remember that SAS keeps track of time in seconds since midnight and that the *seconds* argument in the DHMS function can be greater than 59. Therefore, if you supply the SAS time value as the *seconds* argument and set *hour* and *minute* to zeros, that will work just fine. The following example demonstrates:

```
1  DATA ex5_1;
2  INPUT row $ sasdate :date9. sastime :time8.;
3  result = DHMS(sasdate,0,0,sastime);
4  fmt_result = result;
5  format sasdate date9. sastime timeampm.;
6  DATALINES;
7  A 16oct2015 17:30
8  B 08jun2016 11:00
9  C 14apr2015 00:00
10 ;;;
11 RUN;

12 PROC PRINT DATA=ex5_1 NOOBS LABEL SPLIT='\';
13 ID row;
14 VAR sasdate sastime result fmt_result;
15 FORMAT fmt_result datetime19. sasdate sastime;
16 LABEL row='Row'
17       result = '~{text_align=c}SAS Datetime Value\Calculated
   Using\DHMS(sasdate,0,0,sastime)'
18 fmt_result = "Result Formatted with\DATETIME19. Format";
19 RUN;
```

The Result

Row	sasdate	sastime	SAS Datetime Value Calculated Using DHMS(sasdate,0,0,sastime)	Result Formatted with DATETIME19. Format
A	20377	63000	1760635800	16OCT2015:17:30:00
B	20613	39600	1781002800	08JUN2016:11:00:00
C	20192	0	1744588800	14APR2015:00:00:00

The calculation of the datetime value is done in line 3 of the above program. The formats have been removed from the sasdate and sastime variables in the table for the purpose of illustrating the actual values that are being sent to the function. Row C above has been included to demonstrate that any valid SAS time value will work, even 0 seconds. What good is that? If you need to create a SAS datetime value and you only have a date without a time value, then you can use DHMS(sasdate,0,0,0) to convert your SAS date into a SAS datetime.

HMS(hour,minute,second);

HMS(hour,minute,second); creates a SAS time value. *hour*, *minute*, and *second* are all numeric variables and/or constants. None of the parameters are restricted to their clock times. Therefore, *hour* can be greater than 24, while *minute* and *second* can be greater than 60. All three arguments must be present or you will get a missing value as the result. The following table provides examples:

Sample Function Call	Time Value	Formatted with TIME. Format	Formatted with TIMEAMPM. Format	Comments
HMS(18,0,9)	64809	18:00:09	6:00:09 PM	
HMS(7,45,80)	27980	7:46:20	7:46:20 AM	The time is not displayed as "*7:45:80*" because the value is returned as the total number of seconds, and the format is applied to that. Neither the TIME. nor the TIMEAMPM. formats display minute or second values greater than 59.
HMS(15,03,35.56)	54215.56	15:03:36	3:03:36 PM	
HMS(8,17,33)	29853	8:17:33	8:17:33 AM	
HMS(21,14,28)	76468	21:14:28	9:14:28 PM	

MDY(month,day,year);

MDY(month,day,year); creates a SAS date value from the arguments. All three arguments are required. *month*, *day*, and *year* are all numeric variables or constants. If year is two digits, the century will be determined by the YEARCUTOFF= option. If a value given for any of the arguments is not valid or missing, such as MDY(2,31,2014) (February 31, 2014), the function will return a missing value and give you an "invalid argument to function" message in the log. The following table shows examples:

Sample Function Call	SAS Date Value	Formatted with WEEKDATE. Format
MDY(9,3,1876)	–30434	Sunday, September 3, 1876
MDY(12,14,15)	20436	Monday, December 14, 2015
MDY(3,26,1915)	–16352	Friday, March 26, 1915
MDY(5,22,2014)	19865	Thursday, May 22, 2014

NWKDOM(n, weekday, month, year);

NWKDOM(n, weekday, month, year); creates a SAS date value for a given weekday in a given week number from a given month and year. All of the arguments are numeric and can be represented by constants or numeric variables. If any of the arguments are missing, or not valid, then the function will return a missing value. *n* ranges from 1 (first) to 5 (last). Sometimes using the value of 5 will give the same result as 4, if the fourth week of the month is also the last week of the month. *weekday* ranges from 1 to 7, where 1 represents Sunday, and 7 represents Saturday. *month* can range from 1 (January) to 12 (December). *year* represents the year and is subject to the YEARCUTOFF= option if you use two digits for the year.

You can use this function to find dates that are expressed as "The first Saturday in May," or the "last Tuesday in August," easily. The following table provides examples of how this function works.

Date Description	Sample Function Call	SAS Date Value	Date Formatted using WEEKDATE.
First Tuesday in November.	NWKDOM(1,3,11,2014)	20031	Tuesday, November 4, 2014
Second Tuesday in December.	NWKDOM(2,3,12,2014)	20066	Tuesday, December 9, 2014
Third Wednesday in April.	NWKDOM(3,4,4,2014)	19829	Wednesday, April 16, 2014
Fourth Sunday in May.	NWKDOM(4,1,5,2014)	19868	**Sunday, May 25, 2014**
Last Sunday in May.	*NWKDOM(5,1,5,2014)*	*19868*	***Sunday, May 25, 2014***
First Sunday in June.	NWKDOM(1,1,6,2018)	21338	Sunday, June 3, 2018
Second Saturday in December.	NWKDOM(2,7,12,2018)	21526	Saturday, December 8, 2018
Third Saturday in May.	NWKDOM(3,7,5,2018)	21323	Saturday, May 19, 2018
Fourth Monday in October.	NWKDOM(4,2,10,2018)	21479	**Monday, October 22, 2018**
Last Monday in October.	*NWKDOM(5,2,10,2018)*	*21486*	***Monday, October 29, 2018***

The bolded and italicized text above shows that the last week of the month and the fourth week of the month may or may not produce the same date.

YYQ(year,qtr);

YYQ(year,qtr); creates a SAS date value from the arguments. Both arguments are required. *year* is a numeric variable or constant representing the year, and *qtr* is a numeric variable or constant between 1 and 4, representing the quarter of the year. If year is two digits, the century will be determined by the YEARCUTOFF= system option. This function returns the date of the first day of the quarter in the given year. The following table provides examples of how this function works.

Sample Function Call	SAS Date Value	Formatted with MMDDYY10. Format	Comment
YYQ(2015,1)	20089	01/01/2015	
YYQ(99,3)	14426	07/01/1999	
YYQ(25,2)	–12693	04/01/1925	When YEARCUTOFF=1920, 25 translates to 1925 because the range runs from 1920 through 2019.
YYQ(25,2)	23832	04/01/2025	In SAS 9.4, the YEARCUTOFF= option default is 1926, so the range is now 1926 through 2025, and 25 is now translated as 2025.
YYQ(2015,2)	20179	07/01/2015	

5.4 Calculating Elapsed Time, and the HOLIDAY() Function

Because SAS uses simple math as the basis for dates and times, you might think that calculating elapsed time or projecting into the future would be easy. It should be a matter of simple addition or subtraction. However, SAS provides several functions that deal with calculating elapsed time, and for the most part the function is going to be more accurate than simple math. For example, one of the mathematical equations for calculating age is (current date–date of birth)/365.25. This approximation uses the .25 to account for leap years, but it fails to take into account the exception for years that are divisible by 100 but not by 400. While you might rarely need such accuracy when calculating elapsed years, the efficiency of a SAS function might speed things perceptibly when working with big data.

5.4.1 Calculating Elapsed Time with DATDIF() and YRDIF()

These two functions were originally developed for use with securities calculations for specific financial instruments.

DATDIF(start,end,basis);

DATDIF(start,end,basis); calculates the number of days between two dates. *start* is the starting date, which can be a date constant, a numeric variable, or a SAS expression. *end* is the ending date, also a date constant, a numeric variable, or a SAS expression. *basis* is a character constant or variable that tells SAS how to calculate the difference. The *start* and *end* arguments are required, while *basis* is optional. *basis* has two possible values. Note that if you use a character constant for *basis*, remember that it will need to be enclosed in quotation marks, or you will get an error.

1. **'30/360'**, which sets each month to 30 days, and the year to 360 days, regardless of how many days are in each month or year in the span between the two dates. If a day is at the end of a month (for example, February 28/29 or March 31), it will be considered as the 30th of the month.

2. **'ACT/ACT'**, which uses the actual number of days in each month and year in the span between the two dates. This is the default, and it is identical to subtracting start from end.

Sample Function Call	Result	Comment
DATDIF('19JUL2015'd,'19JUL2016'd,'30/360')	360	*basis* is "30/360," indicating a year of 360 days by definition.
DATDIF('19JUL2015'd,'19JUL2016'd,'ACT/ACT')	366	2016 is a leap year, so 366 days have elapsed between July 19, 2015, and July 19, 2016.

YRDIF(start,end,basis);

YRDIF(start,end,basis); calculates the number of years between two dates. It is almost always more accurate than using mathematical approximation, depending on the basis used and the desired result. *start* is the starting date, which can be a date constant, a numeric variable, or a SAS expression. *end* is the ending date, also a date constant, a numeric variable, or a SAS expression. *basis* is a character constant or variable that tells SAS how to calculate the difference. *basis* has five possible values, as compared with the two possibilities in the DATDIF() function:

1. **'30/360'**, which sets each month to 30 days, and the year to 360 days, regardless of how many days are in each month or year in the span between the two dates. If a day is at the end of a month (for example, February 28/29 or March 31), it will be considered as the 30th of the month.
2. **'ACT/ACT'**, which uses the actual number of days in each month and year in the span between the two dates. This was the default basis through SAS version 9.2. You can use the alias "Actual," not 'ACT.'
3. **'ACT/360'**, which uses the actual number of days between the two dates to calculate the number of years, but it uses a 360-day year, regardless of how many days are in each year, so the result is number of days divided by 360.
4. **'ACT/365'**, which uses the actual number of days between the two dates to calculate the number of years, but uses a 365-day year, regardless of how many days are in each year, so the result is number of days divided by 365.
5. **'AGE'**, which is used to calculate a person's age. This is available starting with SAS version 9.3, and as of that release, 'AGE' is now the default.

Sample Function Call	Resulting value	Comment
YRDIF('07AUG1967'd,'24MAY2014'd,'30/360')	46.797222	A month is defined as having 30 days, and the year is 360 days long.

Sample Function Call	Resulting value	Comment
YRDIF('07AUG1967'd,'24MAY2014'd,'ACT/ACT')	46.794521	Actual days in a month and actual days in a year are used.
YRDIF('07AUG1967'd,'24MAY2014'd,'ACT/360')	47.477778	Actual number of days in a month are used; year is defined as having 360 days,
YRDIF('07AUG1967'd,'24MAY2014'd,'ACT/365')	46.827397	Actual number of days in a month are used; year is defined as having 365 days

As you can see, all four results are different, and this is due to the way they were calculated. Prior to version 9.3, this function was often used to calculate ages, but even the 'ACT/ACT' basis doesn't calculate ages precisely. The 'ACT/ACT' basis averages leap year days across the four years. Now let's examine the YRDIF function when the 'ACT/ACT' basis and the 'AGE' basis are used.

Sample Function Call	Resulting value
YRDIF('07AUG1967'd,'24MAY2014'd,'ACT/ACT')	46.794521
YRDIF('07AUG1967'd,'24MAY2014'd,'AGE')	46.794521

This looks as if the YRDIF() function will yield the same result for both the 'AGE' basis and the 'ACT/ACT' basis. Where is the difference? Let's look at another series of dates.

	Sample Function Call	Resulting value
A	YRDIF('07AUG1968'd,'24MAY2014'd,'ACT/ACT')	45.79342
	YRDIF('07AUG1968'd,'24MAY2014'd,'AGE')	45.794521
B	YRDIF('07AUG1969'd,'24MAY2014'd,'ACT/ACT')	44.794521
	YRDIF('07AUG1969'd,'24MAY2014'd,'AGE')	44.794521
C	YRDIF('07AUG1971'd,'24MAY2014'd,'ACT/ACT')	42.794521
	YRDIF('07AUG1971'd,'24MAY2014'd,'AGE')	42.794521
D	YRDIF('07AUG1972'd,'24MAY2014'd,'ACT/ACT')	41.79342
	YRDIF('07AUG1972'd,'24MAY2014'd,'AGE')	41.794521

In groupings A and D, you see that the basis makes a difference, while in groupings B and C, the 'ACT/ACT' and 'AGE' return identical results. What makes groupings A and D so different? These are leap years, and the leap day is accounted for as a whole day for that given year, as opposed to the averaging of a quarter day per year performed by the 'ACT/ACT' algorithm.

(U.S. and Canada Only) HOLIDAY(holiday,year);

HOLIDAY(holiday,year); provides the date of selected holidays in any given year as a SAS date value. This function is valid for U.S. and Canada holidays only. *holiday* can be a character string enclosed in quotation marks or a character variable containing one of the arguments listed below. The valid values of *holiday* are listed in the table below. Note: If you use a character variable instead of a string, your variable should be at least 18 characters long to accommodate the longest argument.

Argument Used in Function	Holiday	Observed Date
BOXING	Boxing Day	December 26
CANADA	Canada Day	July 1
CANADAOBSERVED	Canada Day observed	July 1, or July 2 if July 1 is a Sunday
CHRISTMAS	Christmas	December 25
COLUMBUS	Columbus Day	2nd Monday in October
EASTER	Easter Sunday	date varies
FATHERS	Father's Day	3rd Sunday in June
HALLOWEEN	Halloween	October 31
LABOR	Labor Day	1st Monday in September
MLK	Martin Luther King, Jr. 's birthday	3rd Monday in January beginning in 1986
MEMORIAL	Memorial Day	last Monday in May (since 1971)
MOTHERS	Mother's Day	2nd Sunday in May
NEWYEAR	New Year's Day	January 1
THANKSGIVING	U.S. Thanksgiving Day	4th Thursday in November
THANKSGIVINGCANADA	Canadian Thanksgiving Day	2nd Monday in October
USINDEPENDENCE	U.S. Independence Day	July 4
USPRESIDENTS	Abraham Lincoln's and George Washington's birthdays observed	3rd Monday in February (since 1971)
VALENTINES	Valentine's Day	February 14
VETERANS	Veterans Day	November 11
VETERANSUSG	Veterans Day (U.S. government-observed)	U.S. government-observed date for Monday–Friday schedule
VETERANSUSPS	Veterans Day (U.S. post office observed)	U.S. government-observed date for Monday–Saturday schedule (U.S. Post Office)

Argument Used in Function	Holiday	Observed Date
VICTORIA	Victoria Day	Monday on or preceding May 24

Sample Function Call	SAS Date Value	Date Formatted using WEEKDATE.
HOLIDAY("EASTER",2014)	19833	Sunday, April 20, 2014
HOLIDAY("EASTER",2019)	21660	Sunday, April 21, 2019
HOLIDAY("EASTER",2026)	24201	Sunday, April 5, 2026
HOLIDAY("EASTER",2039)	28954	Sunday, April 10, 2039
HOLIDAY("THANKSGIVINGCANADA",2014)	20009	Monday, October 13, 2014
HOLIDAY("THANKSGIVINGCANADA",2019)	21836	Monday, October 14, 2019
HOLIDAY("THANKSGIVINGCANADA",2026)	24391	Monday, October 12, 2026
HOLIDAY("THANKSGIVINGCANADA",2039)	29137	Monday, October 10, 2039
HOLIDAY("USINDEPENDENCE",2014)	19908	Friday, July 4, 2014
HOLIDAY("USINDEPENDENCE",2019)	21734	Thursday, July 4, 2019
HOLIDAY("USINDEPENDENCE",2026)	24291	Saturday, July 4, 2026
HOLIDAY("USINDEPENDENCE",2039)	29039	Monday, July 4, 2039

5.5 The Basics of SAS Intervals

Some of the SAS functions described in section 5.4, such as DATDIF, are very good at calculating the exact amount of elapsed time between two SAS dates, and as demonstrated in some of the above examples, you can see the difference between the function and simple math. There aren't functions to project future dates, because it would seem simple enough: you just add a number of days, hours, or minutes, and you come up with an answer.

However, we frequently need to refer to units of time that are not uniform, such as months, which can be 28, 29, 30, or 31 days long. SAS provides functions to calculate intervals because, in many cases, simple math is still only an approximation. SAS has several standard interval definitions that are used with dates, times, and datetimes that represent many of the normal periods of time that we refer to, such as weeks or quarters. You are not restricted to the intervals given in the standard definitions, because you also have the ability to easily modify them. You can use multipliers and/or a shift index in conjunction with the standard intervals. Multipliers enable you to define intervals that are multiples of a standard interval and are not already defined, such as a decade or a century. A shift index enables you to define intervals that do not correspond with the starting values used by an interval (standard OR with a multiplier), such as a fiscal year that begins in July instead of January, or, to give you an example with a multiplier, a decade that starts in years ending in the

number '5,' instead of years ending in 0. Section 5.6 will discuss the concepts of multipliers and the shift index in detail. If you need intervals that cannot be described by using multipliers and/or a shift index with the standard SAS intervals, SAS has the capacity for you to define your own intervals, and this is covered in depth in Section 5.7.

For the remainder of this book, when the term "interval" is used in a function definition, it means a SAS interval name, along with any multiplier and/or shift index unless explicitly specified otherwise. We will begin our discussion of intervals by providing a list of all the standard interval definitions and the periods that they describe in Table 5.5.

Table 5.5 SAS Interval Definitions Used with Dates, Times, and Datetimes

Category	Interval Name	Definition	Default Starting Point
Date	DAY	Daily intervals	Each day
	WEEK	Weekly intervals of seven days	Each Sunday
	WEEKDAYdaysW	Daily intervals with Friday-Saturday-Sunday counted as the same day (five-day work week with a Saturday-Sunday weekend). *days* identifies the individual numbers of the weekend day(s) by number (1=Sunday ... 7=Saturday). By default, days="17," so the default interval is WEEKDAY17W.	Each day
	TENDAY	Ten-day intervals (a U.S. automobile industry convention)	1st, 11th, and 21st of each month
	SEMIMONTH	Half-month intervals	First and sixteenth of each month
	MONTH	Monthly intervals	First of each month
	QTR	Quarterly (three-month) intervals	1-Jan 1-Apr 1-Jul 1-Oct
	SEMIYEAR	Semi-annual (six-month) intervals	1-Jan 1 Jul
	YEAR	Yearly intervals	1-Jan
Datetime	DTDAY	Daily intervals	Each day
	DTWEEK	Weekly intervals of seven days	Each Sunday

Category	Interval Name	Definition	Default Starting Point
	DTWEEKDAYdaysW	Daily intervals with Friday-Saturday-Sunday counted as the same day (five-day work week with a Saturday-Sunday weekend). *days* identifies the individual weekend days by number (1=Sunday ... 7=Saturday). By default, days="17," so the default interval is DTWEEKDAY17W.	Each day
	DTTENDAY	Ten-day intervals (a U.S. automobile industry convention)	1st, 11th, and 21st of each month
	DTSEMIMONTH	Half-month intervals	First and sixteenth of each month
	DTMONTH	Monthly intervals	First of each month
	DTQTR	Quarterly (three-month) intervals	1-Jan 1-Apr 1-Jul 1-Oct
	DTSEMIYEAR	Semiannual (six-month) intervals	1- Jan 1 Jul
	DTYEAR	Yearly intervals	1-Jan
	DTSECOND	Second intervals	Seconds
	DTMINUTE	Minute intervals	Minutes
	DTHOUR	Hour intervals	Hours
Time	SECOND	Second intervals	Seconds
	MINUTE	Minute intervals	Minutes
	HOUR	Hourly intervals	Hours

5.5.1 The Interval Calculation Functions: INTCK() and INTNX()

The interval calculation functions INTCK() and INTNX() use SAS interval definitions. INTCK() counts the number of intervals between two given dates, times, or datetimes. INTNX() calculates the date, time, or datetime that results after a given number of intervals have been added to an initial date, time, or datetime value.

The syntax for the INTCK function is as follows.

INTCK(interval, start-of-period,end-of-period,method);

interval is the SAS designation for a period of time, and can be a character literal or character variable that corresponds to one of the defined time intervals (see Table 5.5). *start-of-period* is the beginning date, time, or datetime value, while *end-of-period* is the ending one. Both *start-of-period* and *end-of-period* can be anything that evaluates to a valid SAS date, time, or datetime value.

As of SAS version 9, *method* determines how SAS is going to count the intervals. There are two possible values: CONTINUOUS (or C or CONT) and DISCRETE or (D or DISC). The default is DISCRETE, and this has been how intervals have traditionally been calculated in SAS. When *method* is DISCRETE, the INTCK() function is counting the number of times that the period *interval* begins between *start-of-period* and *end-of-period*, inclusive. It does not count the number of complete intervals between *start-of-period* and *end-of-period*. This also means that the count does not begin with *start-of-period*, but at the beginning of the first interval after that. The following example demonstrates how INTCK() counts using the DISCRETE method. For example, take the dates Saturday, December 31, 2011, and Sunday, January 1, 2012.

Example 5.2: How the INTCK() Function Counts by Default (*method* Is DISCRETE)

Function Call	Result
INTCK('DAY','31dec2011'd,'01jan2012'd);	1
INTCK('WEEK','31dec2011'd,'01jan2012'd)	1
INTCK('MONTH','31dec2011'd,'01jan2012'd)	1
INTCK('YEAR','31dec2011'd,'01jan2012'd)	1

All of the intervals are equal to 1 even though only one day has passed! January 1, 2012, is the start of day, week, month, and year intervals. The starting day occurred on December 31, and the ending day began on January 1, so it is obvious that the result for the DAY interval should be 1, because the DAY interval boundary was crossed on January 1. However, when it comes to the WEEK interval, Sunday is the beginning of the week. Therefore, the week containing December 31st started on Sunday, December 25. Sunday, January 1, is the beginning of the next week, so you cross the WEEK interval boundary at January 1. This means that one WEEK interval has elapsed between the start of the week in December and the start of the week in January, so that causes the result for WEEK to be 1. Similarly, the month containing December 31 started on December 1, 2011, and the month for January 1, 2012, started on January 1. You are crossing the MONTH interval boundary on January 1, so one MONTH interval has elapsed between the start of the intervals for the two dates, and therefore, that result is 1 as well. Finally, the year for December 31, 2011, started on January 1 of 2011, while the year for January 1, 2012, starts on the same date. You cross the YEAR interval boundary at January 1, which causes the YEAR interval result to be 1. All of the values are equal to 1 because the INTCK() function is counting the DAY, WEEK, MONTH, and YEAR interval boundary, which occurs at Sunday, January 1, 2012, and not because of the number of days, weeks, months, or years that have passed.

Now let's look at the corresponding results when you use CONTINUOUS for the method.

Example 5.3: How the INTCK() Function Counts When *method* Is CONTINUOUS

Function Call	Result
INTCK('DAY','31dec2011'd,'01jan2012'd,'C');	1
INTCK('WEEK','31dec2011'd,'01jan2012'd,'C')	0
INTCK('MONTH','31dec2011'd,'01jan2012'd,'C'	0
INTCK('YEAR','31dec2011'd,'01jan2012'd,'C')	0

One day has passed, as in the DISCRETE example. However, this looks more like what you might expect when you ask how many weeks, months, and years have passed between December 31, 2011, and January 1, 2012. Only one day has passed, not an entire week, month, or year. The CONTINUOUS method calculates continuous time, and it uses the calendar definition of a week, month, or year, starting on the date supplied as the first argument. Let's change the ending dates appropriately to reproduce the results in Example 5.2.

Example 5.4: How the INTCK() Function Counts When *method* Is CONTINUOUS

Function Call	Result	Days Elapsed
INTCK('DAY','31dec2011'd,'01jan2012'd,'C');	1	**1**
INTCK('WEEK','31dec2011'd,**'07jan2012'd,**'C')	1	**7**
INTCK('MONTH','31dec2011'd,**'31jan2012'd**,'C'	1	**31**
INTCK('YEAR','31dec2011'd,**'31dec2012'd**,'C')	1	**366**

Now the picture should be a little more clear that using CONTINUOUS causes the INTCK() function to look at continuous elapsed time, not interval boundaries. To firm up this demonstration of the CONTINUOUS method for the INTCK() function, let's look at how the definition of the MONTH interval can change.

Example 5.5: How the INTCK() Function Counts When *method* Is CONTINUOUS

	Function Call	Result	Days Elapsed
A	INTCK('MONTH','01jan2012'd,'01feb2012'd,'C');	1	31
B	INTCK('MONTH','01jan2012'd,'31jan2012'd,'C');	0	30
C	INTCK('MONTH','01apr2012'd,'01may2012'd,'C');	1	30
D	INTCK('MONTH','28feb2012'd,'28mar2012'd,'C');	1	29
E	INTCK('MONTH','29feb2012'd,'28mar2012'd,'C');	0	28
F	INTCK('MONTH','28feb2013'd,'28mar2013'd,'C');	1	28

Rows A and B look normal: 31 days is a month, while 30 days is not a month. However, row C states that 30 days is a month. This is only true for the months of April, June, September, and November. Finally, rows D through F demonstrate that, when you are using the CONTINUOUS method, the MONTH interval changes based on the month and the year. In leap years, a period of 29 days across February is considered a month (row D). However, a period of 28 days is not considered a month in leap years (row E). On the other hand, if it is not a leap year, 28 days in February is a month.

The CONTINUOUS method is useful for calculating anniversaries and milestones tied to dates, times, and datetimes. Although this method seems much more intuitive than the traditional (and still default) way that SAS handles intervals, you will have to be very cautious in choosing which method you use and when. It is not recommended that you replace all of your existing code to use this method instead of the default, because the two methods differ in their definition of interval boundaries. The CONTINUOUS method shifts the interval by the starting date provided. With the DISCRETE method, all values within an interval boundary are considered to be equivalent. That makes it possible to group observations occurring within an interval for analysis.

Since the CONTINUOUS method is more intuitive to the way we tend to look at periods of time, the remainder of this section on the INTCK() function will be devoted to the traditional DISCRETE method. Please remember that the DISCRETE method is the default, and in order to use the CONTINUOUS method, you will have to supply the proper method argument in the INTCK() function. All of the following examples rely on the default method, so there is no method argument used in the function calls.

To complete the picture of how the traditional use of the INTCK() function works, let's look at the effect that the ending date has on INTCK().

Example 5.6: How the INTCK() Function Counts

Function Call	Result
INTCK('DAY','31dec2014'd,'06jan2015'd);	6
INTCK('WEEK','31dec2014'd,'06jan2015'd)	1
INTCK('MONTH','31dec2014'd,'06jan2015'd	1
INTCK('YEAR','31dec2014'd,'06jan2015'd)	1

Here you can see that although 6 days have elapsed, still only 1 week, month, and year have elapsed according to INTCK()! That is because the start of the week, month, and year interval for January 6, 2015, is still January 1, 2015, and that is what INTCK() is counting. Here are some more examples of the use of the INTCK() function with its default of the DISCRETE method.

Example 5.7: The INTCK Function: The Basics

Function Call	Result
INTCK('DAY','15jun2013'd,'22jun2013'd);	7
INTCK('WEEK','01jan2016'd,'01jan2017'd);	53
INTCK('DTDAY','01oct1872:08:00:00'dt,'20dec1872:18:00:00'dt);	80
INTCK('MONTH','05mar2000'd,'01may2000'd);	2

Example 5.8: The INTCK Function: Counting Backward

Function Call	Result
INTCK('YEAR','22dec2015'd,'16jul2010'd);	-5

In this example, the from date is after the to date, and therefore, the answer is negative. Since INTCK() counts interval boundaries, the answer is –5 because it starts counting at the start of a year. The start of the year interval for December 22, 2015, is January 1, 2015, so it is not counted. January 1, 2014; January 1, 2013; January 1, 2012; January 1, 2011; and January 1, 2010, are the boundaries of the YEAR intervals that it is counting, which corresponds to the beginning dates of each year.

Example 5.9: The INTCK Function: Counting Weekdays

	Function Call	Result
A	INTCK('WEEKDAY17W','12JAN2014'd,'14JAN2014'd);	2
B	INTCK('WEEKDAY17W','13JAN2014'd,'14JAN2014'd);	1
C	INTCK('WEEKDAY17W','17JAN2014'd,'18JAN2014'd);	0

If you are counting the number of work weekdays that have elapsed, you must be careful to remember that INTCK() counts interval boundaries and that the starting date is not counted in the answer. Row A above seems perfectly reasonable. You would expect that there would be two weekdays between Sunday, January 12, 2014, and Tuesday, January 14, 2014. Even though the starting boundary for WEEKDAY17W is Monday, the starting date is Sunday, January 12, 2014, so it counts Monday and Tuesday. However, you might think that there are two weekdays in the span Monday, January 13, 2014, to Tuesday, January 14, 2014. In row B, the INTCK() function is counting only 1 weekday, because it doesn't include Monday, January 13, 2014 (the start argument used in the function), when it counts the interval boundaries. Why is row C equal to zero then? First, Friday is the starting day, and the starting day is never included in the count. January 18, 2014, is a Saturday, and since we've defined the weekend days as Saturday and Sunday, those days are outside of any interval boundaries, so by definition, no interval boundaries have been passed.

Example 5.10 illustrates the difference between the three methods that have been discussed for calculating elapsed years using SAS. We will use the YRDIF() function using both the 'AGE' basis

and 'ACT/ACT' basis. Since the INTCK() function also counts elapsed intervals, we will use it and show both the DISCRETE and CONTINUOUS methods, and compare those with mathematical estimation.

Example 5.10: The YRDIF() Function as Opposed to Mathematical Estimation and INTCK()

This uses the same dates, August 7, 1963, and May 8, 2014, for all five methods, but row A uses the YRDIF() function with 'ACT/ACT,' row B uses the same function, but with the 'AGE' basis. The two values are equivalent with the YRDIF() function, but as we have seen earlier, 'AGE' and 'ACT/ACT' only yield different results when the year being tested is a leap year. The mathematical approximation divides the number of days between the two dates by 365.25, and although the discrepancy is minute in this example, the difference is caused by the fact that the number of leap years in the period (11) is not evenly divisible by 4, rendering the value 365.25 an approximation. The INTCK function using the DISCRETE method counts interval boundaries from their beginning, so it is counting the number of January firsts between January 1, 1963, and January 1, 2015. In effect, unless you were born on January 1, using the DISCRETE method with INTCK() to calculate your age will make you old before your time! It is more appropriate in this type of scenario to use the CONTINUOUS method with the INTCK() function, which yields the answer 50 years old.

	Calculation method	Result
A	Using YRDIF() with actual days in month, actual year	50.750685
B	Using YRDIF() with AGE basis	50.750685
C	Using ('08may2014'd - '07aug1963'd)/365.25	50.75154
D	Using INTCK('YEAR','07aug1963'd,'08may2014'd,'DISCRETE')	51
E	Using INTCK('YEAR','07aug1963'd,'08may2014'd,'CONTINUOUS')	50

Those are some examples of the use of the INTCK() function. Remember, as of SAS version 9, the INTCK() function has an additional argument that you can use to change the way that INTCK counts interval boundaries. However, any existing code using INTCK() does not need to be changed, because the default method is the 'DISCRETE' method, which is how INTCK() has always worked. Once again, it is not a good idea to change existing code without considering how INTCK is being used, because the methods differ significantly in their calculations.

The next of the interval functions that we will consider is the INTNX() function. This function takes a given SAS date, time, or datetime value and calculates a new value by incrementing by the given number of intervals. Where INTCK() calculates the number of intervals between any two date, time, or datetime values, INTNX() takes the start of the period and increments it by a number of intervals to give the end of the period. The syntax of the INTNX() function is as follows:

INTNX(interval,start-from,number-of-increments,alignment);

INTNX(interval,start-from,number-of-increments,alignment); is one of the SAS intervals defined in Table 5.5 and can be a character literal or character variable that evaluates to one of the defined intervals. *start-from* is the starting date, time, or datetime value, which can be a constant, numeric variable, or a SAS expression. *number-of-increments* is an integer constant or a numeric variable that indicates how many intervals to advance. If it is not an integer, only the integer portion of the value will be used. *alignment* sets the returned date, time, or datetime value according to one of four predefined settings. The function calculates the dates at the beginning of the interval period, and then the alignment argument adjusts the result. The values are: Beginning or B, Middle or M, End or E, and Same or S (Sameday is also acceptable as an alias). The default value for alignment is Beginning. The Sameday argument cannot be used with the DTQTR, DTSEMIYEAR, or DTYEAR intervals, and it has no effect on time values.

The next series of examples demonstrates various uses and effects of the INTNX() function. The first example is the default use of the function with each of the date, time, and datetime intervals, while the second shows what happens when you use a non-integer as the increment to the function. Example 5.14 illustrates the use of the alignment arguments to yield specific dates.

Example 5.11: The INTNX() Function with Default Alignment

Each of the examples below increments the date, time, or datetime value by 3 of the interval shown in bold italics. For dates and datetimes, the start date is the same: Thursday, November 6, 2014. The only difference is that datetime intervals (interval names starting with DT) return datetime values, not dates. The interval values for datetime values are calculated in seconds, not days. By default, the INTNX() function returns the beginning of the interval given. If you want to change this, use the alignment argument discussed above.

Function Call	Result	Comment
INTNX('DAY','06NOV2014'd,3)	November 9, 2014	
INTNX('DTDAY','06NOV2014:15:00:00'dt,3)	11/9/2014 12:00 AM	The time returned is midnight of 11/09/2014, not 3 p.m.
INTNX('WEEKDAY17W','06NOV2014'd,3)	November 11, 2014	The returned date is the following Tuesday. Saturday (7) and Sunday (1) don't count since they are defined as the weekend days.
INTNX('DTWEEKDAY17W','06NOV2014:15:00:00'dt,3)	11/11/2014 12:00 AM	
INTNX('WEEK','06NOV2014'd,3)	November 23, 2014	The value returned is the Sunday of the week, not 21 calendar days.
INTNX('DTWEEK','06NOV2014:15:00:00'dt,3)	11/23/2014 12:00 AM	

Function Call	Result	Comment
INTNX('TENDAY','06NOV2014'd,3)	December 1, 2014	The value returned is the first of the month, 25 calendar days, not 30. This interval will always return either the 1st, 11th, or 21st of the month.
INTNX('DTTENDAY','06NOV2014:15:00:00'dt,3)	12/1/2014 12:00 AM	
INTNX('SEMIMONTH','06NOV2014'd,3)	December 16, 2014	Although November has 30 days, the returned date is the 16th, not 45 calendar days, which would be the 21st.
INTNX('DTSEMIMONTH','06NOV2014:15:00:00'dt,3)	12/16/2014 12:00 AM	
INTNX('MONTH','06NOV2014'd,3)	February 1, 2015	The date returned is the beginning of the following month, not the same date. To get the same date, use the "S" (Sameday) alignment argument.
INTNX('DTMONTH','06NOV2014:15:00:00'dt,3)	2/1/2015 12:00 AM	
INTNX('QTR','06NOV2014'd,3)	July 1, 2015	The first day of the quarter is returned.
INTNX('DTQTR','06NOV2014:15:00:00'dt,3)	7/1/2015 12:00 AM	
INTNX('SEMIYEAR','06NOV2014'd,3)	January 1, 2016	The beginning of the 3rd semi-year after 11/02/2014 is January 1, 2016.
INTNX('DTSEMIYEAR','06NOV2014:15:00:00'dt,3)	1/1/2016 12:00 AM	
INTNX('YEAR','06NOV2014'd,3)	January 1, 2017	The beginning of the 3rd year after 11/02/2014 is January 1, 2017.
INTNX('DTYEAR','06NOV2014:15:00:00'dt,3)	1/1/2017 12:00 AM	
INTNX('SECOND','8:00:00 AM't,3)	8:00:03 AM	
INTNX('MINUTE','8:00:00 AM't,3)	8:03:00 AM	
INTNX('HOUR','8:00:00 AM't,3)	11:00:00 AM	

Example 5.12: The INTNX() Function: Using Non-Integer Increments

INTNX('HOUR','16:45't,2.5)	06:00 PM (18:00)

When the number of increments argument is not an integer, SAS will take the integer part as the value. In this case, 2.5 becomes 2. The beginning of the first hour increment is 17:00, and the

second is 18:00. Since alignment is the default value of B or beginning, that sets the answer to the beginning of the hour, which gives the result of 18:00.

Example 5.13: Counting Backward with the INTNX() Function

INTNX('DAY','29NOV2014'd,-4)	25NOV2014

INTNX() always adds the number of intervals to the starting date, time, or datetime. If you want to find a prior date, then use a negative increment.

Example 5.14: The INTNX() Function with Alignment Arguments

INTNX Call	Date	Comment
INTNX('WEEK','02NOV2014'd,3,'**B**')	11/23/2014	Beginning of the week interval. Sunday, November 23, 2014
INTNX('WEEK','02NOV2014'd,3,'**M**')	11/26/2014	Middle of the week interval. Wednesday, November 26, 2014
INTNX('WEEK','02NOV2014'd,3,'**E**')	11/29/2014	End of the week interval. Saturday, November 29, 2014
INTNX('WEEK','02NOV2014'd,3,'**S**')	11/23/2014	Same day, so the week interval ends on the same day of the week (Sunday), leaving the duration at 21 days. Sunday, November 23, 2014

Although the alignment arguments do change the value returned by the INTNX() function, it must be reiterated that the function performs the calculation based on the beginning of the interval, and the adjustment to the final result only occurs after that first calculation has been done.

5.6 Modifying SAS Intervals

The default SAS intervals don't match every situation. However, there are two ways to customize SAS intervals: You can modify the existing SAS intervals with multipliers and a shift index, or you can create your own intervals from a SAS data set. This section will discuss multipliers and the shift index. Section 5.7 will discuss creating your own intervals.

The interval function INTCK(*interval,start-of-period,end-of-period,method*) counts the number of intervals between two given SAS dates, times, or datetimes, while the INTNX(*interval,start-from,number-of-increments,alignment*) function advances a given date, time, or datetime by a specified number of intervals. Both functions use the standard SAS interval definitions. (See Table 5.6, below.) With the DISCRETE method, the INTCK() function counts at the starting point of a given interval, while the CONTINUOUS method adjusts the starting point of the interval based on the starting date and counts from there. Each method has its merits, and which one you use will depend on your analysis or reporting needs.

The INTNX() function always advances to the beginning of the given interval. The *alignment* argument available with INTNX() will adjust the *result* to the middle, end, or the same day of the interval. Nonetheless, the intervals are still measured from their starting point, and the adjustment is only applied *after* INTNX() has moved the specified number of intervals.

This behavior can lead to problems when your definition of an interval doesn't exactly match the standard SAS definition of that same interval. For example, the standard SAS definition of year has the first day of the year defined as starting January 1. What happens if your fiscal year starts on July 1, and that is how you want to measure your year? What if your semi-month period is truly 14 days long?

A shift index and interval multipliers enable you to do this. You have the ability to define the start of an interval definition by adding a shift index to it. The shift index defines the number of shift points to move the start of the interval. There is one restriction on the value of the shift index: It must be less than the number of shift points within the interval. For example, you can shift the start of a YEAR interval by up to 12 months, but not 13, as there are only 12 months in a year. This makes sense as if you were to shift a year by 13 months, you have shifted into the next YEAR interval, and this is much less intuitive than adding 1 year to the starting date(s) that you send to the interval function.

Table 5.6 shows the SAS interval name, definition, and the shift increment period for each one.

Table 5.6: SAS Intervals and Their Shift Points

Category	Interval	Definition	Shift Point
Date	DAY	Daily intervals	Days
	WEEK	Weekly intervals of seven days	Days
	WEEKDAY<daysW>	Daily intervals with Friday-Saturday-Sunday counted as the same day. *days* identifies the individual weekend days by number (1=Sunday ... 7=Saturday). By default, days="17," so the default interval is WEEKDAY17W.	Days
	TENDAY	Ten-day intervals	Ten-day periods
	SEMIMONTH	Half-month intervals	Semimonthly periods
	MONTH	Monthly intervals	Months
	QTR	Quarterly (three-month) intervals	Months
	SEMIYEAR	Semi-annual (six-month) intervals	Months
	YEAR	Yearly intervals	Months
Datetime	DTDAY	Daily intervals	Days

Category	Interval	Definition	Shift Point
	DTWEEK	Weekly intervals of seven days	Days
	DTWEEKDAY<daysW>	Daily intervals with Friday-Saturday-Sunday counted as the same day. *days* identifies the individual weekend days by number (1=Sunday ... 7=Saturday). By default, days="17," so the default interval is DTWEEKDAY17W.	Days
	DTTENDAY	Ten-day intervals	Ten-day periods
	DTSEMIMONTH	Half-month intervals	Semimonthly periods
	DTMONTH	Monthly intervals	Months
	DTQTR	Quarterly (three-month) intervals	Months
	DTSEMIYEAR	Semiannual (six-month) intervals	Months
	DTYEAR	Yearly intervals	Months
	DTSECOND	Second intervals	Seconds
	DTMINUTE	Minute intervals	Minutes
	DTHOUR	Hour intervals	Hours
Time	SECOND	Second intervals	Seconds
	MINUTE	Minute intervals	Minutes
	HOUR	Hourly intervals	Hours

For our first example, let's use the standard case of a fiscal year that starts on July 1. Technically, a shift index says that you are moving the start of the interval to the beginning of one of the subperiods within that interval. Years are subdivided into months, so that is the shift unit. The start point is always included in the count of subperiods to be shifted. The easiest way for me to remember this is that shifting an interval by one subperiod is the same as the unshifted base interval. Think of this another way: while the start of the year is January 1, the beginning of the first month in the year is also January 1. If we use a shift index of 1 with our YEAR interval, we are shifting to the beginning of the first month in the year.

Therefore, in order to move your YEAR interval by seven months, you need to move it to the beginning of the seventh subperiod. (January 1, February 1, March 1, April 1, May 1, June 1, July 1—that's seven). The shift value is added to the interval name by appending it with a decimal point.

Therefore, in order to advance the start of the YEAR interval by seven months to July 1, you would use the interval "YEAR.7," as illustrated by the following:

Example 5.15: Moving the Start of a Year Interval from January 1 to July 1

	Sample Function Call	Result
A	INTCK('YEAR','01jan2013'd,'06jul2014'd,'DISCRETE');	1
B	INTCK('YEAR','01jan2013'd,'06jul2014'd,'CONTINUOUS');	1
C	INTCK('YEAR.7','01jan2013'd,'06jul2014'd,'DISCRETE');	2
D	INTCK('YEAR.7','01jan2013'd,'06jul2014'd,'CONTINUOUS');	1

INTCK() counts the start of interval boundaries with the DISCRETE method (rows A and C). When the interval is "YEAR," it is measuring from January 1, 2013, until January 1, 2014 (the start of the year containing July 6, 2014). When the interval is "YEAR.7," you have shifted the beginning of the YEAR interval by 7 months, which declares that the year starts on July 1. In the example above, that shift causes the measurement to begin on July 1, 2012 (the start of the shifted "year" containing January 1, 2013) and end on July 1, 2014, which is the start of the year containing July 6, 2004. That is why the value returned in row C is 2, not 1. With the CONTINUOUS method, the shifted year starts on July 1, 2013, so the start date is before the start of the measurement. Therefore, the only interval boundary crossed is on July 1, 2014, so the answer in row D is 1.

While a shift index enables you to move the starting point of any given SAS interval, an interval multiplier enables you to define the length of your own intervals. You define your new interval by applying a multiplier value to an interval. For example, if you want to measure biweekly (14-day) periods, you would take the WEEK interval and multiply it by 2. You add the numeric multiplier to the end of the SAS interval name, so in this case your interval name would be "WEEK2." It measures 14-day periods starting on Sunday because the regular SAS interval WEEK starts on Sunday. Example 5.16 demonstrates the creation and use of a custom interval, by using the example of a multiplier of 2 for the WEEK interval. It also illustrates some of the features of custom intervals.

Example 5.16: Using an Interval Multiplier to Create a Custom Interval

Sample Function Call	Result
INTNX('WEEK','09feb2014'd,2,'S');	Sunday, February 23, 2014
INTNX('WEEK2','09feb2014'd,2,'S');	Sunday, March 9, 2014

In the above example, the start of the WEEK interval two weeks from February 9 is February 23, so it makes sense that the start of the second biweekly period from February 9 is March 9, because February 2014 has 28 days, which is the same as two of your WEEK2 intervals. This is not as

simple as it seems. When using SAS intervals with the INTNX() function, you must keep in mind that intervals are always measured from the beginning of the starting interval to the beginning of the ending interval. This is independent of the starting and ending dates that you supply. Let's move the starting date in Example 5.16 backward by one week to February 2, 2014.

	Sample Function Call	Result
1	INTNX('WEEK','02feb2014'd,2,'S');	Sunday, February 16, 2014
2	INTNX('WEEK2','02feb2014'd,2,'S');	Sunday, February 23, 2014

Advancing by two WEEKS (row 1) is still a difference of 14 days, as expected. What happened in row 2? Shouldn't you get 28 days instead of 21? No, because the start of the WEEK2 interval containing February 2, 2014, is January 24, 2014, and that is where the INTNX() function begins its count. Here is a sample program to produce the starting dates of the intervals:

Example 5.17: WEEK2 Intervals from January 1, 2014

```
DATA tricky;
DO intervals= 0 TO 5;
   interval_start = INTNX('WEEK2','01jan2014'd,intervals);
   OUTPUT;
END;
RUN;

PROC REPORT DATA=tricky NOWD SPLIT='\';
COLUMNS intervals interval_start;
DEFINE intervals / DISPLAY "Number of WEEK2\intervals from\January
1, 2014"
       STYLE={CELLWIDTH=1.25in TEXT_ALIGN=C};
DEFINE interval_start / FORMAT=weekdate32. "Starting Date of
Interval";
RUN;
```

The above program creates a table that shows the starting dates for the first five WEEK2 intervals of 2014. If the date(s) supplied fall between the starting dates of any two boundaries, the interval count (or incrementing) will begin from the starting date of the previous interval. This is why moving the starting date of the INTNX function by one week doesn't necessarily move the result by one week. You can move up to 14 days within a WEEK2 interval before you change the interval boundary.

Number of WEEK2 intervals from January 1, 2014	Starting Date of Interval
0	Sunday, December 29, 2013

Number of WEEK2 intervals from January 1, 2014	Starting Date of Interval
1	Sunday, January 12, 2014
2	Sunday, January 26, 2014
3	Sunday, February 9, 2014
4	Sunday, February 23, 2014
5	Sunday, March 9, 2014

How does SAS determine what the starting date of a given interval is if you use a multiplier? It takes the multiplied interval that you've created and starts counting beginning with January 1, 1960. This is true for all multiplied intervals except multiplied WEEK intervals. Multiplied WEEK intervals are counted starting from Sunday, December 27, 1959, because weeks are defined as starting on Sundays, and January 1, 1960, was a Friday.

You can also use a multiplier and a shift indicator together if needed. Interval multipliers are directly appended to the interval name (for example, WEEK2), and shift indicators are appended to the interval name with a leading decimal point (for example, WEEK**.**5). To use both the multiplier and a shift index, you first append the multiplier to the interval name to create a new interval name (for example, WEEK2), and then you append the shift indicator to the interval name. Given a multiplier of 2, and a shift of 5, the interval name becomes "WEEK2.5." Again, you cannot use a shift index that is greater than the number of shift points in the interval. In the case of a WEEK2 interval, you would not be able to use a shift index greater than 14, as that would place the starting point of your interval into the next WEEK2 interval. In that case, it would be more intuitive to change the starting date(s) that you will supply to the interval function.

To demonstrate, let's expand on the scenario used in example 5.17. What if you wanted your biweekly periods to start on January 1, 2014? January 1, 2014, was a Wednesday, so you want to move the starting date to the fourth day of the week (Sunday=1, therefore, Wednesday=4). Now we'll generate the same table as above, using a shift indicator of 4 days in addition to the biweekly interval of WEEK2.

Example 5.18: Using Both an Interval Multiplier and Shift Index to Create a Custom Interval

```
DATA tricky;
DO intervals= 0 TO 5;
   plain_interval_start = INTNX('WEEK2','01jan2014'd,intervals);
   shifted_interval_start = INTNX('WEEK2.4','01jan2014'd,intervals);
   OUTPUT;
END;
RUN;

PROC REPORT DATA=tricky NOWD SPLIT='\';
COLUMNS intervals plain_interval_start shifted_interval_start;
DEFINE intervals / "Number of WEEK2\intervals from\January 1, 2014"
       STYLE={CELLWIDTH=1.25in TEXT_ALIGN=C} DISPLAY;
DEFINE plain_interval_start /. "Starting Date of WEEK2 Interval"
       FORMAT=weekdate32.;
DEFINE shifted_interval_start / "Starting Date of WEEK2.4 Interval"
       FORMAT=weekdate32.;
RUN;
```

Number of WEEK2 Intervals from January 1, 2014	Starting Date of WEEK2 Interval	Starting Date of WEEK2.4 Interval
0	Sunday, December 29, 2013	Wednesday, January 1, 2014
1	Sunday, January 12, 2014	Wednesday, January 15, 2014
2	Sunday, January 26, 2014	Wednesday, January 29, 2014
3	Sunday, February 9, 2014	Wednesday, February 12, 2014
4	Sunday, February 23, 2014	Wednesday, February 26, 2014
5	Sunday, March 9, 2014	Wednesday, March 12, 2014

When you use a multiplier, you also have the ability to define your shifts within the entire interval created by the multiplier. For example, let's create a decade interval by using YEAR10 as the interval. Remember that intervals start at the beginning of the boundary, so the decades would start at the beginning of the first year of the decade. What can you do if you want to define the decade as starting in May of the middle year in the decade (for example, May of 1955 as opposed to January of 1950)?

To shift intervals across years, you need to use the first nested interval within the YEAR interval, which is MONTH. So you would use (*number of years to shift**12) to calculate the number of months that you need to shift. If you wanted to shift 5 years, you would use 5*12=60. Add 5 to that, which shifts the starting month from January to May, and your interval definition is now

YEAR10.65. That would be decades starting in May of years that end in 5. The code below shows the effect of moving the interval by 65 months.

Example 5.19: Shifting a Multiplied Interval

```
DATA tricky3;
DO intervals= 0 TO 7;
   plain_interval_start = INTNX('YEAR10','01sep1950'd,intervals);
   shifted_interval_start =
INTNX('YEAR10.65','01sep1950'd,intervals);
   OUTPUT;
END;
RUN;

ods rtf file="c:\book\2ndEd\examples\ex5.6.5.rtf";

PROC REPORT DATA=tricky3 NOWD SPLIT='\';
COLUMNS intervals plain_interval_start shifted_interval_start;
DEFINE intervals / DISPLAY "Number of\intervals from\September 1,
1950"
       STYLE={CELLWIDTH=1in TEXT_ALIGN=C};
DEFINE plain_interval_start / FORMAT=weekdate32.
       "Starting Date of YEAR10 Interval";
DEFINE shifted_interval_start / FORMAT=weekdate32.
       "Starting Date of Shifted YEAR10 Interval";
RUN;
```

Number of Intervals from September 1, 1950	**Starting Date of YEAR10 Interval**	**Starting Date of Shifted YEAR10 Interval**
0	Sunday, January 1, 1950	***Tuesday, May 1, 1945***
1	Friday, January 1, 1960	Sunday, May 1, 1955
2	Thursday, January 1, 1970	Saturday, May 1, 1965
3	Tuesday, January 1, 1980	Thursday, May 1, 1975
4	Monday, January 1, 1990	Wednesday, May 1, 1985
5	Saturday, January 1, 2000	Monday, May 1, 1995
6	Friday, January 1, 2010	Sunday, May 1, 2005
7	Wednesday, January 1, 2020	Friday, May 1, 2015

The first thing to note is that even though the date that we specified is September 1, the starting date of the interval is January 1, because that is the start of the YEAR interval. In the second

column, you can see that the interval has been shifted. Even though the starting date of the YEAR10. interval is January 1, 1950, the shifted interval itself starts on Tuesday, May 1, 1945 (bold italics added), not May 1, 1955. Why? Because it is the start of the interval that contains January 1, 1950.

The next example will demonstrate the importance of making sure that you use the correct shift index and that you have to be aware of the starting date and/or datetime for your shifted, multiplied interval.

Example 5.20: The Interaction between the Starting Date and the Interval Starting Point: The "Working Shift" Interval

A company has a 24-hour production cycle that consists of three 8-hour working shifts per day. It is easy to create an interval that represents each shift by using HOUR8 as your base interval, but shifts do not start at midnight, 8 a.m., and 4 p.m., which would be the default for the HOUR8 interval. The table below shows when the HOUR8 interval starts based on the default.

Function Call	Starting Time
INTNX('HOUR8','00:00't,0)	12:00:00 AM
INTNX('HOUR8','00:00't,1)	8:00:00 AM
INTNX('HOUR8','00:00't,2)	4:00:00 PM

However, in this case, the actual shifts start at 6 a.m., 2 p.m., and 10 p.m., so we will need to use a shift index to align our HOUR8 interval with the actual shift hours. It is important to remember that when you are shifting intervals, a shift index of 1 moves the start of the interval to the start of the first subperiod within the interval—which is always the same as the start of the unshifted interval. Look at the following table, which will tell us what interval shift will give us the starting point at 6 a.m.

Function Call	Interval Name	Starting Time	Comment
INTNX('HOUR8','6:00't,0);	HOUR8	12:00 AM	
INTNX(HOUR8.1,'6:00't,0)	HOUR8.1	12:00 AM	Same as HOUR8 interval.
INTNX(HOUR8.2,'6:00't,0)	HOUR8.2	1:00 AM	
INTNX(HOUR8.3,'6:00't,0)	HOUR8.3	2:00 AM	
INTNX(HOUR8.4,'6:00't,0)	HOUR8.4	3:00 AM	
INTNX(HOUR8.5,'6:00't,0)	HOUR8.5	4:00 AM	

Function Call	Interval Name	Starting Time	Comment
INTNX(HOUR8.6,'6:00't,0)	HOUR8.6	5:00 AM	
INTNX(HOUR8.7,'6:00't,0)	**HOUR8.7**	6:00 AM	This is what we want.

Notice that in the above table, we've used 6 a.m. as our reference time in the INTNX() function call, and we get the starting point of the interval by adding 0 intervals to our reference point. This method works for finding the starting point of any interval in relation to the reference point. Now we'll repeat our first table, showing the starting points for a 24-hour period starting at 6 a.m., with our HOUR8.7 interval to demonstrate that everything is correct.

Function Call	Starting Time
INTNX('HOUR8.7','06:00't,0)	6:00:00 AM
INTNX('HOUR8.7','06:00't,1)	2:00:00 PM
INTNX('HOUR8.7','06:00't,2)	10:00:00 PM

Although it might seem odd that a shift index of 7 is needed to get a change of 6 hours, you must remember that the shift index is the number of subperiods within the interval to move. The shift is always to the beginning of each subperiod within the interval. Therefore, the first subperiod within an interval always starts at the beginning of the interval. In this case, the HOUR8 interval starts at midnight, and the first hour within the HOUR8 interval also starts at midnight. That is why the shift index necessary is 7, not 6.

In summary, the most important thing to remember about using intervals, multipliers, and the shift index is that all intervals, no matter how they are defined, are measured from their beginning. If you are using the INTCK() function, the definition of "beginning" is dependent upon the method that you use, DISCRETE or CONTINUOUS. With DISCRETE, it is considered to be the beginning of the interval boundary, but with CONTINUOUS, beginning is defined based on the start date provided. The INTNX() function measures the beginning from the interval boundary; if it is easier to think of INTNX as always using the DISCRETE method, then you can do that. However, remember that the alignment arguments "B", "M", "E", and "S" for the INTNX() function do not adjust the date until after the function has executed and calculated its start-of-interval result (also remember that the alignment arguments only work with the INTNX() function). No matter what, you can use the interval multipliers and a shift index to move the starting point of an interval, and you can use them anywhere that you can use a date, time, or datetime interval.

5.7 Creating Your Own SAS Intervals

Another way to create intervals for use with interval functions is by creating a SAS data set that defines the boundaries of your interval. This data set defines the duration of periods within the interval and the period across which the interval applies as well, which is an important concept that will be discussed when it comes to errors with using user-defined intervals. User-defined intervals enable you to create intervals that do not correspond with shifted and/or multiplied SAS intervals, such as intervals with irregular boundaries, those that do not encompass an entire predefined SAS interval, or intervals that are specific to your job, company, or industry. However, if you needed to count working shifts in a situation where there are two 10-hour work shifts per day with a four-hour break in-between while the machinery resets, you could not do it by using multipliers and a shift index.

You can also redefine an interval. For example, the shift point for a day is the day; what happens if you want your "day" to start at 6 a.m. instead of midnight? DTHOUR24.7 would create an interval that runs 24 hours starting at 6 a.m. (See example 5.20, which does the same for an HOUR8 interval. The only difference is that there are 24-hour-long subperiods in a DTHOUR24 interval.) Unfortunately, the original question was to shift a DAY interval by 6 hours, but you can't do that with a date value. DTHOUR intervals only apply to datetime values. You would have to convert all of your date data to datetime data in order to use that interval. Defining your own interval where days start at the desired time would solve this problem.

The SAS data set you have to create consists of at least one variable named *begin*. It can also have two other optional variables, *end* and *season*. If you do not include the *end* variable, it is assumed to be one less than the start of the next period within that interval; for example, if your user-defined interval is based on days, the end will be the day before the begin value for the next record. While seasonality is a time series concept outside of the scope of this book, it can be accounted for with the variable *season*. One additional requirement is that *begin* (and *end*, if present in the data set) must have an appropriate date, time, or datetime format assigned at the time that the data set is created. In order to associate this data set with a user-defined interval, you need to use the INTERVALDS system option.

OPTIONS INTERVALDS=(*interval-name*$_1$*, dataset-with-interval-records*$_1$*, interval-name*$_2$*, dataset-with-interval-records*$_2$*...)*

This is where you name your user-defined interval and tell SAS which data set to use for that interval. You can specify as many user-defined intervals as you need in a single OPTIONS INTERVALDS statement. *interval-name* must conform to standard SAS naming conventions, and it cannot be a SAS reserved word. If you try to use a reserved word, you will get an error at the OPTIONS statement. This applies to names of existing intervals defined by SAS as well. You cannot change the definition of an existing SAS interval by creating one of your own with the same name. *dataset-with-interval-records* is a standard SAS data set reference of the type *libref.dataset-name*. Since this statement merely identifies a data set for association with your interval name, data set options are not allowed.

For our first example, let's define a SEMESTER interval, based on an academic calendar with two semesters and a summer session. The sessions are defined as follows: The fall semester begins the day following the first Monday in September and ends on the third Friday in December. The spring semester begins on the second Monday in January, unless that day falls on the Martin Luther King, Jr., holiday. In that case, it begins on the day following the holiday. The summer session starts on the last Tuesday in May and proceeds through the third Friday in August. The following example uses both the NWKDOM() and HOLIDAY() functions described in sections 5.3 and 5.4, respectively.

Example 5.21: Academic Calendar: Creating a SEMESTER Interval

```
OPTIONS INTERVALDS=(semester=semester); ❶

DATA semester;
DO year=2014 TO 2017; ❷

   /* Fall Semester */
   begin = NWKDOM(5,2,8,year); /* Last Monday in August */
   end = NWKDOM(3,6,12,year);  /* Third Friday in December */
   OUTPUT;

   /* Spring Semester */
   begin = NWKDOM(2,2,1,year+1); /* Second Monday in January of
                                    following calendar year */
   if begin eq HOLIDAY('MLK',year+1) then /* If MLK day, then
Tuesday */
      begin = begin + 1;
   end = NWKDOM(2,6,5,year+1); /*  Second Friday in May */
   OUTPUT;

   /* Summer Session */
   begin = NWKDOM(1,2,6,year+1); /* First Monday in June */
   end = NWKDOM(3,6,8,year+1); /* Third Friday in August */
   OUTPUT;
END;
FORMAT begin end WEEKDATE.; /* Required */
RUN;
```

❶ tells SAS that we are defining an interval named "semester," and it is to be based on the data set WORK.SEMESTER.

❷ sets a definite limit on the interval, from the start of the 2014 academic year through the 2017 academic year, which ends in August 2018. Below is the resulting data set:

begin	end
Monday, August 25, 2014	Friday, December 19, 2014
Monday, January 12, 2015	Friday, May 8, 2015

begin	end
Monday, June 1, 2015	Friday, August 14, 2015
Monday, August 31, 2015	Friday, December 18, 2015
Monday, January 11, 2016	Friday, May 13, 2016
Monday, June 6, 2016	Friday, August 12, 2016
Monday, August 29, 2016	Friday, December 16, 2016
Monday, January 9, 2017	Friday, May 12, 2017
Monday, June 5, 2017	Friday, August 11, 2017
Monday, August 28, 2017	Friday, December 15, 2017
Monday, January 8, 2018	Friday, May 11, 2018
Monday, June 4, 2018	Friday, August 10, 2018

You can then use the SEMESTER interval that you have created in the INTCK or INTNX interval functions. The following table shows how many semesters have elapsed from September 15, 2014, to the respective end dates shown.

Example 5.22: Using a User-Defined Interval with INTCK

Function Call	Result
INTCK('SEMESTER',"15SEP2014"d,"22JUN2015"d);	2
INTCK('SEMESTER',"15SEP2014"d,"24NOV2016"d);	6
INTCK('SEMESTER',"15SEP2014"d,"01MAY2018"d);	10

For a second example, let's consider a case where we need to count the actual number of working days for each year. The WEEKDAY interval is a useful approximation, but it doesn't account for holidays, so if you need to know the exact number of working days between two dates, this will solve that problem.

Example 5.23: Customized Company Working Days

```
OPTIONS INTERVALDS=(WorkingDays=Workdays);
DATA workdays (KEEP=begin); /* End variable not needed */
   start = '01JAN2014'D;
   stop  = '31DEC2015'D;
   nweekdays = INTCK('WEEKDAY',start,stop); /* Based on Mon-Fri
weekdays */
   DO i = 0 TO nweekdays;
      begin = INTNX('WEEKDAY',start,i);
      year = YEAR(begin);
```

```
   /* Company-specific holidays */

/* Company closes day after thanksgiving */
       xthanks = HOLIDAY("THANKSGIVING",year) + 1;
/* Christmas Eve */
       xmaseve = HOLIDAY('CHRISTMAS',year) - 1;
/* Friday before Easter */
       sprng = HOLIDAY("EASTER",year) - 2;
/* Founders Day Company Holiday. If on weekend, move forward or back
*/
       founders = MDY(8,6,year);
       SELECT(WEEKDAY(founders));
         WHEN(6) founders = founders - 1;
         WHEN(1) founders = founders + 1;
         OTHERWISE founders = founders;
       END;

      /* Exclude dates of standard and company holidays from
interval data set */
       IF BEGIN ne HOLIDAY("NEWYEAR",year) AND
          BEGIN ne HOLIDAY("MLK",year) AND
          BEGIN ne HOLIDAY("USPRESIDENTS",year) AND
          BEGIN ne HOLIDAY("MEMORIAL",year) AND
          BEGIN ne HOLIDAY("USINDEPENDENCE",year) AND
          BEGIN ne HOLIDAY("LABOR",year) AND
          BEGIN ne HOLIDAY("VETERANS",year) AND
          BEGIN ne HOLIDAY("THANKSGIVING",year) AND
          BEGIN ne HOLIDAY("CHRISTMAS",year) AND
          BEGIN ne xmaseve AND
          BEGIN ne xthanks AND
          BEGIN ne sprng AND
          BEGIN ne founders THEN OUTPUT;
   END;
   FORMAT begin DATE9.;
RUN;
```

The following code generates a table showing the difference between the number of calendar days, the number of weekdays, and the number of working days for this company for the calendar year 2014.

```
DATA CountDays;
   start = '01JAN2014'D;
   stop  = '01JAN2015'D;
   ActualDays = INTCK('DAYS',start,stop);
   Weekdays   = INTCK('WEEKDAYS',start,stop);
   ProductionDays = INTCK('WORKINGDAYS',start,stop);
   LABEL ActualDays="Actual Days"
         Weekdays="Weekdays"
         ProductionDays = "Production Days"
         start = "Starting Date"
```

```
         end = "Ending Date"
   ;
   FORMAT start stop DATE9.;
RUN;

PROC PRINT DATA=CountDays NOOBS;
RUN;
```

start	stop	ActualDays	Weekdays	ProductionDays
01JAN2014	01JAN2015	365	261	248

How much of a difference can this make? Consider that this company has six product lines and the time to manufacture each product varies. A customer comes in with a large order and wants to know what the delivery date would be for each product. It is critical that the delivery be guaranteed to the day, so the contract calls for a substantial discount if the delivery date is not met. In the following table, the order date is June 27, 2014. We use the INTNX() function to add the corresponding number of production days for each product to the order date. The approximate delivery date is calculated using the SAS WEEKDAY interval, while the true delivery date uses our user-defined WORKINGDAYS interval.

```
DATA production_lines;
LENGTH product $ 40;
   orderDate = "27JUN2014"d;
   Product = "Std Product 1";
   days_from_order = 23;
   OUTPUT;
   Product = "Std Product 2";
   days_from_order = 32;
   OUTPUT;
   Product = "Std Product 3";
   days_from_order = 35;
   OUTPUT;
   Product = "Custom Product 1";
   days_from_order = 33;
   OUTPUT;
   Product = "Custom Product 2";
   days_from_order = 42;
   OUTPUT;
   Product = "Custom Product 3";
   days_from_order = 56;
   OUTPUT;
   FORMAT orderdate weekdate.;
RUN;

DATA ordertime;
SET production_lines;
deliveryDate = INTNX('WORKINGDAYS',orderdate,days_from_order,'S');
```

```
approx_deliveryDate =
INTNX('WEEKDAY',orderdate,days_from_order,'S');
FORMAT deliverydate approx_deliveryDate weekdate.;
RUN;

PROC REPORT DATA=ordertime NOWD SPLIT='\';
COLUMNS product days_from_order approx_deliveryDate deliveryDate;
DEFINE product / "Product";
DEFINE days_from_order / "Production Days\Required"
       STYLE={TEXT_ALIGN=C};
DEFINE approx_deliveryDate / STYLE={TEXT_ALIGN=L}
   "~S={text_align=c}Approximate Delivery Date\using WEEKDAYS
Interval";
DEFINE deliveryDate / "~S={TEXT_ALIGN=C}Actual Delivery Date\using
Custom\WORKINGDAYS Interval" STYLE={TEXT_ALIGN=L};
RUN;
```

Product	Production Days Required	Approximate Delivery Date Using WEEKDAYS Interval	Actual Delivery Date Using Custom WORKINGDAYS Interval
Std Product 1	23	Wednesday, July 30, 2014	Thursday, July 31, 2014
Std Product 2	32	Tuesday, August 12, 2014	Thursday, August 14, 2014
Std Product 3	35	Friday, August 15, 2014	Tuesday, August 19, 2014
Custom Product 1	33	Wednesday, August 13, 2014	Friday, August 15, 2014
Custom Product 2	42	Tuesday, August 26, 2014	Thursday, August 28, 2014
Custom Product 3	56	Monday, September 15, 2014	Thursday, September 18, 2014

There is one specific problem to be aware of when you are working with user-defined intervals. If a specific date, time, or datetime is not covered in the interval data set, and you try to use one of these values in a function, or one of these values is to be returned by a function, you will not get the answer that you expect. Such unexpected results can be avoided by ensuring that the dates that you define in the data set encompass all the dates that you may encounter. The next two examples will demonstrate what happens when you use a date that is out of the range of your user-defined interval. We will use the WORKINGDAYS interval as defined in the previous example. That interval is only defined for a period of 2 years, from January 1, 2014, through December 31, 2015. What happens if you try to calculate the number of working days using the INTCK() function with ending dates outside of that range, or you try to project a date using the INTNX() function that would be beyond that range?

Example 5.24: Out-of-Interval Calculation Using INTCK() and "WORKINGDAYS" Custom Interval

Obs	startDate	endDate	result
1	Wednesday, December 23, 2015	Thursday, December 24, 2015	0
2	Wednesday, December 23, 2015	Friday, December 25, 2015	0
3	Wednesday, December 23, 2015	Saturday, December 26, 2015	0
4	Wednesday, December 23, 2015	Sunday, December 27, 2015	0
5	Wednesday, December 23, 2015	Monday, December 28, 2015	1
6	Wednesday, December 23, 2015	Tuesday, December 29, 2015	2
7	Wednesday, December 23, 2015	Wednesday, December 30, 2015	3
8	Wednesday, December 23, 2015	Thursday, December 31, 2015	4
9	Wednesday, December 23, 2015	Friday, January 1, 2016	4
10	Wednesday, December 23, 2015	Saturday, January 2, 2016	4
11	Wednesday, December 23, 2015	Sunday, January 3, 2016	4
12	Wednesday, December 23, 2015	Monday, January 4, 2016	4
13	Wednesday, December 23, 2015	Tuesday, January 5, 2016	4
14	Wednesday, December 23, 2015	Wednesday, January 6, 2016	4
15	Wednesday, December 23, 2015	Thursday, January 7, 2016	4

Rows 1 through 4 make sense. There are no working days from Thursday through Sunday, as December 24 is a company holiday. Rows 5 through 8 count a working day for Monday through Thursday as you would expect. Friday, January 1, 2016, is also a company holiday. Therefore, you would not expect it to count as a working day, nor would you expect the following Saturday and Sunday to count as workdays. But what about that first full week in January, rows 12 through 15? Those days aren't company holidays, so why aren't they being counted as working days?

The answer is that they aren't part of the WORKINGDAYS interval. The definition of the interval ended on December 31, 2015. Because they are not in the data set, any dates before or after the dates specified in the data set will not be considered working days. This makes sense when you consider how we initially created the company holidays. We did not include those dates in the data set that created the interval. Therefore, this problem actually starts with January 1, 2016. It is merely coincidental that January 1st is a holiday, and the 2nd and 3rd are weekend days; that is not what is preventing them from being counted as working days. Their absence from the data set we

used to create the interval is the only reason that they are not being counted. This next example shows the type of problem that can arise when using the INTNX() function in a similar fashion.

Example 5.25: Out-of-Interval Calculation with INTNX() and "WORKINGDAYS" Custom Interval

Obs	Function Call	Result
1	INTNX('workingdays','26dec2015'd,1,'S');	Monday, December 28, 2015
2	INTNX('workingdays','26dec2015'd,2,'S');	Tuesday, December 29, 2015
3	INTNX('workingdays','26dec2015'd,3,'S');	Wednesday, December 30, 2015
4	INTNX('workingdays','26dec2015'd,4,'S');	Thursday, December 31, 2015
5	INTNX('workingdays','26dec2015'd,5,'S');	.
6	INTNX('workingdays','26dec2015'd,6,'S');	.
7	INTNX('workingdays','26dec2015'd,7,'S');	.
8	INTNX('workingdays','26dec2015'd,8,'S');	.

What happened here? Rows 1 through 4 count the working days as expected, Monday through Thursday. However, if you try to increment the starting date by more than 4 working days, the function call returns a missing value. SAS cannot do the calculation because the result is undefined; it ran out of possible results to return at December 31, 2015, because you did not provide any in the data set.

The ability to define your own intervals will cover situations when the combination of standard SAS intervals, a multiplier, and a shift index does not suffice. It is not difficult to create the data set upon which SAS will base your user-defined interval, but be aware that dates that are not included in the interval can cause problems when you try to use the intervals that you create. If you see unexpected results from using your user-defined interval, the first thing to investigate is to make sure that the range that you have defined for your interval encompasses the range of possible results.

5.8 Interval Functions about Intervals

SAS has a series of functions that provide information about intervals that are being used. They can tell you which interval is the best fit between two or three dates, or suggest a format for a given interval. There is a function to tell you the shift point in effect for a given interval. (For basic intervals, this returns the values in Table 5.5, but interval multipliers can change the returned value.) There is also a function that will tell you if the interval name that you are using is valid. This can allow for increased automation and building robust applications.

5.8.1 INTFIT(argument-1,argument-2,type)

INTFIT(*argument-1,argument-2,type*) takes two date, datetime, or observation numbers as the first two arguments and returns the interval between them deemed to be "the best fit." This function assumes that the alignment is "SAME," so that the exact values are matched between the two arguments. *type* can be one of three case-insensitive values:

Interval Name	Definition
D	Use when the two arguments are SAS date values. These can be date literals or variables containing SAS date values
DT	Use when the two arguments are SAS datetime values. These can be datetime literals or variables containing SAS datetime values
OBS	Use this when you want to find the interval between two observations by providing observation numbers. This will return an interval with the base name of "OBS."

You can use this function to give you the interval between two points in time. The interval returned might contain a multiplier and/or a shift index. As you can see in the table below, some of the intervals are not as you might anticipate.

	start	end	result
A	Tuesday, April 1, 2014	Wednesday, April 2, 2014	DAY
B	Tuesday, April 1, 2014	Thursday, April 3, 2014	DAY2
C	Tuesday, April 1, 2014	Friday, April 4, 2014	DAY3.3
D	Tuesday, April 1, 2014	Saturday, April 5, 2014	DAY4.3
E	*Tuesday, April 1, 2014*	*Sunday, April 6, 2014*	*DAY5.5*
F	Tuesday, April 1, 2014	Tuesday, April 8, 2014	WEEK.3
G	Tuesday, April 1, 2014	Wednesday, April 9, 2014	DAY8.7
H	Tuesday, April 1, 2014	Friday, April 11, 2014	TENDAY
I	Tuesday, April 1, 2014	Sunday, April 13, 2014	DAY12.3
J	Tuesday, April 1, 2014	Tuesday, April 15, 2014	WEEK2.10
K	Tuesday, April 1, 2014	Wednesday, April 16, 2014	SEMIMONTH
L	Tuesday, April 1, 2014	Thursday, April 17, 2014	DAY16.7
M	Tuesday, April 1, 2014	Saturday, April 19, 2014	DAY18.15
N	Tuesday, April 1, 2014	Monday, April 21, 2014	TENDAY2.2
O	5/1/2014 9:00 AM	5/1/2014 2:00 PM	DTHOUR5
P	*5/1/2014 9:00 AM*	*5/1/2014 8:45 PM*	*DTMINUTE705.136*

	start	end	result
Q	*5/1/2014 9:00 AM*	*5/2/2014 3:30 AM*	*DTMINUTE1110.61*

The italicized rows show examples of intervals that can be puzzling. Remember that the function provides the exact intervals between the pair of dates or datetimes, and the lowest common denominator will be used for base interval. In addition, the dates are measured using the sameday alignment, which means that the interval is measured from the beginning of the interval, and then adjusted. At first glance, the DAY5.5 result in row E doesn't seem to be right; Tuesday is only the third day of the week, so shouldn't the shift be 3 as it is for the DAY3 and DAY4 intervals? The start of DAY 5 intervals are Sunday, then Friday. If you want to measure a DAY5 interval to a Sunday from a Tuesday, then you will have to measure from the Friday, which gives you a shift of 5. Rows P and Q demonstrate that it might become even more confusing with times and/or datetimes, as the base interval shift points become minutes and/or seconds, so the resulting multipliers and shift indices might be large enough so that you lose the context of the reference point.

The following table shows a sample result of using the INTFIT() function with the 'OBS' type. The records are from a sequential data set with Wednesday, April 2, 2014, as the first date. Since the dates are keyed by observation, the result of the INTFIT() function shows the interval as the relationship between observations, not the actual date values themselves.

Sample Function Call	Date	Result
INTFIT(1,1,'OBS')	Wednesday, April 2, 2014	
INTFIT(1,4,'OBS')	Saturday, April 5, 2014	OBS3.2
INTFIT(1,5,'OBS')	Sunday, April 6, 2014	OBS4.2
INTFIT(1,10,'OBS')	Friday, April 11, 2014	OBS9.2
INTFIT(1,12,'OBS')	Sunday, April 13, 2014	OBS11.2
INTFIT(1,13,'OBS')	Monday, April 14, 2014	OBS12.2
INTFIT(1,14,'OBS')	Tuesday, April 15, 2014	OBS13.2
INTFIT(1,16,'OBS')	Thursday, April 17, 2014	OBS15.2
INTFIT(1,19,'OBS')	Sunday, April 20, 2014	OBS18.2

5.8.2 INTFMT('interval','size')

INTFMT(*'interval','size'*) takes the interval that you give and returns a suggested format for date, time, or datetime values using this interval. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired), or a user-defined interval that you have created, as shown in Section 5.6. If *interval* is represented by a character string, then it must be enclosed in quotation marks, but you can use a character variable containing the name of an interval instead. *size* refers to whether the format returned will have a two-digit year ('short'/'s') or a four-digit year ('long'/'l'). This argument must be enclosed in quotation marks also, but as with the interval

argument, you can use a character variable for the argument. There is one important usage note about the *size* argument: If you do not use lowercase, the function might return an unpredictable result (see Row A) in the example table below:

	Sample Function Call	Result	Comment
A	INTFMT('WEEK','LONG')		***size*** argument is in uppercase
B	INTFMT('WEEK','long')	WEEKDATX17.	
C	INTFMT('MONTH','long')	MONYY7.	
D	INTFMT('QTR','short')	YYQC4.	YYQC. format width is only 4 because it displays two-digit year when the *size* argument='short'
E	INTFMT('QTR','long')	YYQC6.	
F	INTFMT('YEAR','long')	YEAR4.	
G	INTFMT('YEAR10','l')	YEAR4.	Using alias 'l' for 'long'
H	INTFMT('YEAR100','l')	YEAR4.	Using alias 'l' for 'long'
I	INTFMT('DTMONTH','l')	DTMONYY7.	Using alias 'l' for 'long'
J	INTFMT('DTYEAR','l')	DTYEAR4.	Using alias 'l' for 'long'
K	INTFMT('HOUR','l')	DATETIME10.	Using alias 'l' for 'long'
L	INTFMT('WORKINGDAYS','l')	DATE11.	Using alias 'l' for 'long'

This function is most useful when you use the INTFIT or INTGET functions to determine the interval dynamically, so that you will always display a result in the proper context. See example 5.23 to see how this can be used to dynamically format results.

5.8.3 INTGET(argument1,argument2,argument3)

INTGET(*argument1,argument2,argument3*) determines an interval from three date or datetime values that you provide. The arguments can be variables or date/datetime literals, but they must be of the same type; you cannot mix dates and datetimes as arguments. The function calculates all intervals between the first two arguments, and then between the second and third arguments. If the intervals are the same, it will return that interval. If the intervals are not the same, then the function will test the interval between the second and third arguments to see whether it is a multiple of the interval between the first two arguments. If this is true, then the function will return the smaller of the two intervals. If neither of these cases are true, then the INTGET() function will return a missing value.

	Sample Function Call	Result
A	INTGET('05SEP2013'd,20SEP2013'd,05OCT2013'd)	SEMIMONTH
B	INTGET('15JAN2014'd,15APR2014'd,15OCT2015'd)	QTR
C	***INTGET('15JAN2014'd,15APR2015'd,15OCT2015'd)***	
D	INTGET('09JUL2015'd,09SEP2015'd,09MAR2016'd)	MONTH2
E	INTGET('09JUL2015'd,29AUG2015'd,15SEP2015'd)	DAY17.15

In row A, the period between the dates is 15 days, which corresponds to the SEMIMONTH interval. Row B demonstrates that the dates do not have to have the same interval. The gap between January and April is one quarter, but the gap between April and October of the following year is five quarters. Since the interval for the second pair of arguments would be 'QTR5,' since it is a multiple of the QTR interval, the function returns the unmultiplied interval. What happened in row C? Simply, the interval between January 2014 and April 2015 is now QTR5, but the interval between April and October 2015 is QTR2. Since QTR2 is not a multiple of QTR5, the result is missing. Row D shows that if the interval between two arguments is itself a multiplied interval (MONTH2 versus MONTH6), the function will return the interval with the smallest multiplier, while row E shows the ingenuity of the function: The DAY17 interval works if it is shifted by 15 days.

5.8.4 INTSHIFT('interval')

INTSHIFT('*interval*') takes the interval that you give and returns the shift point for that interval. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired). If *interval* is represented by a character string, then it must be enclosed in quotation marks, but you can use a character variable containing the name of an interval instead. If you try to use this function with a user-defined interval that you have created, the function will return a missing value.

Sample Function Call	Result	Comment
INTSHIFT('WEEK')	DAY	
INTSHIFT('MONTH')	MONTH	
INTSHIFT('QTR3.4')	MONTH	
INTSHIFT('YEAR')	MONTH	
INTSHIFT('YEAR10.5')	MONTH	
INTSHIFT('YEAR100')	MONTH	
INTSHIFT('DTMONTH')	DTMONTH	
INTSHIFT('DTYEAR.7')	DTMONTH	
INTSHIFT('HOUR')	DTHOUR	Note that shift points for time intervals are expressed as datetime shift points.

Sample Function Call	Result	Comment
INTSHIFT('WORKINGDAYS')		User-defined interval, so no shift points can be calculated.

5.8.5 INTTEST('interval')

INTTEST('*interval*') takes the interval that you give and returns a 1 if it is a valid interval name, or 0 if it is not. You can use this to determine whether you have created a valid interval with multipliers and/or a shift index. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired). If *interval* is represented by a character string, then it must be enclosed in quotation marks, but you can use a character variable containing the name of an interval instead. If you try to use this function with a user-defined interval that you have created, the function will return a missing value.

Sample Function Call	Result	Comment
INTTEST('WEEK')	1	
INTTEST('QTR.1')	1	You can shift a quarter by up to 3 months (counting the initial starting point as .1).
INTTEST('QTR3.13')	0	The shift point for quarters is months, so the maximum number of months that you can shift in a 3-quarter period would be 9.
INTTEST('YEAR.7')	1	
INTTEST('YEAR10.14')	1	
INTTEST('DTMONTH')	1	
INTTEST('HOUR2.3')	0	Cannot shift 3 hours in a 2-hour interval.
INTTEST('WORKINGDAYS')	0	This function does not recognize user-defined intervals.

5.9 Retail Calendar Intervals and Seasonality

This section covers intervals and functions that are most frequently used with time series analyses. Although the topic of time series analysis is beyond the scope of this book, these intervals and functions are described here because the intervals can be used with INTNX() and INTCK(), while the functions are a part of Base SAS. However, they are presented without context.

5.9.1 Retail Calendar Intervals

SAS has added intervals that are specifically designed for the retail industry. These intervals are ISO 8601 compliant, and can be used with any of the interval functions. They facilitate comparisons across years, because the weeks remain consistent between years. In order to facilitate this, some years will have leap weeks. Year definitions are based on the ISO 8601 definition of a week, which is the first Monday preceding January 4, which in some cases might place the

beginning of the week in December. These intervals enable you to define the structure of your 52-week year, expecting that for the first 13-week period of your interval, there will be 1 month that has 5 weeks in it. This means that you can set the month pattern to 5-4-4, 4-5-4, or 4-4-5. You can work with years, months, or quarters in this fashion. The full list of retail intervals is given in Table 5.7.

Table 5.7: Retail Calendar Intervals

Interval	Description
YEARV	Specifies ISO 8601 yearly intervals. The ISO 8601 year begins on the Monday on or immediately preceding January 4. Note that it is possible for the ISO 8601 year to begin in December of the preceding year. Also, some ISO 8601 years contain a leap week. The beginning subperiod *s* is written in ISO 8601 weeks (WEEKV).
R445YR	Is the same as YEARV except that in the retail industry the beginning subperiod *s* is 4-4-5 months (R445MON).
R454YR	Is the same as YEARV except that in the retail industry the beginning subperiod *s* is 4-5-4 months (R454MON).
R544YR	Is the same as YEARV except that in the retail industry the beginning subperiod *s* is 5-4-4 months (R544MON).
R445QTR	Specifies retail 4-4-5 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod *s* is 4-4-5 months (R445MON).
R454QTR	Specifies retail 4-5-4 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod *s* is 4-5-4 months (R454MON).
R544QTR	Specifies retail 5-4-4 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod *s* is 5-4-4 months (R544MON).
R445MON	Specifies retail 4-4-5 monthly intervals. The 3rd, 6th, 9th, and 12th months are five ISO 8601 weeks long with the exception that some 12th months contain leap weeks. All other months are four ISO 8601 weeks long. R445MON intervals begin with the 1st, 5th, 9th, 14th, 18th, 22nd, 27th, 31st, 35th, 40th, 44th, and 48th weeks of the ISO year. The beginning subperiod *s* is 4-4-5 months (R445MON).
R454MON	Specifies retail 4-5-4 monthly intervals. The 2nd, 5th, 8th, and 11th months are five ISO 8601 weeks long with the exception that some 12th months contain leap weeks. R454MON intervals begin with the 1st, 5th, 10th, 14th, 18th, 23rd, 27th, 31st, 36th, 40th, 44th, and 49th weeks of the ISO year. The beginning subperiod *s* is 4-5-4 months (R454MON).
R544MON	Specifies retail 5-4-4 monthly intervals. The 1st, 4th, 7th, and 10th months are five ISO 8601 weeks long. All other months are four ISO 8601 weeks long with the exception that some 12th months contain leap weeks. R544MON intervals begin with the 1st, 6th, 10th, 14th, 19th, 23rd, 27th, 32nd, 36th, 40th, 45th, and 49th weeks of the ISO year. The beginning subperiod *s* is 5-4-4 months (R544MON).

Interval	Description
WEEKV	Specifies ISO 8601 weekly intervals of seven days. Each week begins on Monday. The beginning subperiod s is calculated in days (DAY). Note that WEEKV differs from WEEK in that WEEKV.1 begins on Monday, WEEKV.2 begins on Tuesday, and so on.

5.9.2 Seasonality Functions

Seasonality is used in time series analysis and can be used in SAS/ETS. It helps to account for normal seasonal variations in patterns inside an analysis. While it is not strictly a date and time matter, it does use intervals. Therefore, the seasonality functions are documented here, but without context. For more information about seasonality and its application, you can refer to the SAS/ETS documentation, and support.sas.com is a great place to find more help on this topic.

INTCINDEX('interval',argument)

INTCINDEX(*'interval',argument*) returns the index of the seasonal cycle based on *interval* for *argument*, where *argument* is a SAS date, time, or datetime value. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired).

INTCYCLE('interval',seasonality)

INTCYCLE(*'interval',seasonality*) returns the interval of the seasonal cycle. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired). *seasonality* is an optional argument that enables you to define seasonal cycles, and it can be a number or a cycle (such as 'YEAR'). For example, you can use the *seasonality* argument to specify your year as having 53 weeks instead of 52.

INTINDEX('interval',argument,seasonality)

INTINDEX(*'interval',argument,seasonality*) returns the seasonal index when given an interval, a SAS date, time, or datetime of the seasonal cycle. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired. *argument* is a SAS date, time, or datetime value. Note that the interval specified must be appropriate for the *argument*. *seasonality* is an optional argument that enables you to define seasonal cycles, and it can be a number or a cycle (such as 'YEAR'). The INTINDEX function returns the seasonal index, while the INTCINDEX function returns the cycle index.

INTSEAS('interval',seasonality)

INTSEAS(*'interval',seasonality*) returns the number of intervals in a seasonal cycle. *interval* can be any standard SAS interval (with multipliers and/or a shift index if desired). *seasonality* is an optional argument that enables you to define seasonal cycles. This is a good function to be aware of if you are not familiar with concepts of seasonality, because the number of intervals in a seasonal cycle might not be intuitive. For example, while you might expect the number of intervals for the QTR interval to be 4 (because there are 4 quarters in a year), the number of intervals for a DAY interval is 7, for the number of days in the week. This can help you check your expectations.

Chapter 6 Deeper into Dates and Times with SAS

6.1 Macro Variables and Dates 185
6.2 Graphing Dates 194
6.3 The Basics of PROC EXPAND 200
6.4 International Date, Time, and Datetime Formats and Informats 212
6.5 NLS Date, Time, and Datetime Conversion Functions 221
6.6 Date Formats and Informats for Other Calendars 225
6.7 Other Software and Their Dates (Excel, Oracle, DB2) 227
6.8 Conclusion 229

6.1 Macro Variables and Dates

There is a high potential for confusion when it comes to the subject of macro variables and dates. Although you have access to dates and times in the SAS macro language with the automatic macro variables &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME, the display of these values is fixed, and therefore they do not give you the power of SAS formats, or of SAS date and time functions.

6.1.1 Automatic Macro Variables

None of these automatic macro variables can be assigned values with %LET or CALL SYMPUT(), or by any other means. They are read-only and work by reading the operating system clock when a SAS session is started. This means that they do not change within a given SAS session.

&SYSDATE

&SYSDATE cannot be modified and displays the system date as a SAS date value formatted with the DATE7. format. If the date according to the operating system when the SAS session starts is July 17, 2014, then &SYSDATE would be equal to "17JUL14."

&SYSDATE9

&SYSDATE9 is identical to &SYSDATE, except that it formats the system date with the DATE9. format and is therefore Y2K-compliant. If the operating system date is November 22, 2015, then &SYSDATE9 would be equal to "22NOV2015."

&SYSDAY

&SYSDAY displays the name of the day that the SAS session began (according to the operating system). If you started a SAS job or session on Monday, April 14, 2014, &SYSDAY would be equal to "Monday."

&SYSTIME

&SYSTIME displays the system time (according to the operating system) in TIME5. format. If the system time is 3:36 p.m. when you start SAS, then &SYSTIME would be equal to "15:36."

6.1.2 Putting Dates into Titles

One of the prime uses of dates in macro variables is for custom titles and footnotes. To use a macro variable in a title, you just need to enclose the TITLE or FOOTNOTE statement that contains the macro variable with double quotation marks like this:

```
TITLE "Today's Date is &SYSDATE9";
```

To see this in context, look at the following program:

Example 6.1: Using Automatic Macro Variables in a Title

```
TITLE "This is how to put a macro variable in a title: Today's Date is
&SYSDATE9";
ODS RTF FILE="examples\title.rtf";
PROC PRINT DATA=sashelp.company (OBS=5);
ID job1;
VAR depthead;
RUN;
ODS RTF CLOSE;
```

Here is the resulting output:

1

This is how to put a macro variable in a title: Today's Date is 28JUN2014

NAME	pop
BAHAMAS	323,063
BELIZE	269,736
CANADA	32,268,243
COSTA RICA	4,327,228
CUBA	11,269,400

While the automatic macro variables &SYSDATE, &SYSDATE9, &SYSDAY, and &SYSTIME might give you the information that you need, their formats are fixed and cannot be changed. Given the multiple ways that dates, times, and datetimes can be displayed, it would be good if you could use the formats and functions to get the display of dates and times that you want in your titles.

6.1.3 Using %SYSFUNC() to Create Dates, Times, and Datetimes in Macro Variables

Many of the date and time functions, as well as formats, are available in the SAS macro language through the %QSYSFUNC() and %SYSFUNC() macro functions. When you are using one of these macro functions with dates and times, you can use two arguments: The first is the date, time, or datetime function that you wish to use. The second argument is optional, but you can specify the format that should be applied to the result.

Example 6.2: Date Functions with %SYSFUNC()

This example puts the formatted value of today's date into the macro variable &DATE and demonstrates the difference between unformatted and formatted dates in macro variables. The date value in &RAWDATE is stored in macro space as a text string and will need to be converted if you want to use it for any calculations. &DATE is also a text string in macro space, but it shows that the WORDDATE. format has been applied to the value.

```
%LET rawdate=%SYSFUNC(DATE());
%LET date=%SYSFUNC(DATE(),WORDDATE32.);

TITLE "Today's SAS Date Value is &rawdate";
TITLE2 "Formatted date: &date";

ODS RTF FILE="c:\book\2ndEd\examples\title2.rtf";
PROC PRINT DATA=sashelp.class (OBS=5);
ID name;
VAR age;
RUN;
ODS RTF CLOSE;
```

Now you can use the result in a title or footnote.

1

Today's SAS Date Value is 19902
Formatted date: June 28, 2014

Name	Age
Alfred	14
Alice	13
Barbara	13
Carol	14
Henry	14

You can also use this as a part of an output file name or Excel worksheet name.

Example 6.3: Putting a Date Value into an Output File Name

```
%LET filedate=%SYSFUNC(DATE(),yymmddd10.);

ODS RTF FILE="Car_Report_&filedate..rtf";
PROC PRINT DATA=sashelp.cars NOOBS LABEL;
RUN;
ODS RTF CLOSE;
```

When you try this at home, you will see that you have generated an RTF file named "Car_report_yyyy-mm-dd.rtf."

Example 6.4: Naming Worksheets with the Date

In this example, we create a custom format to display dates as the full month name and 4-digit year and apply it to a copy of the date variable in the data set. Then we use that variable as the BY variable in the PRINT procedure. Not only does this put the name on each worksheet, but it also filters the dates into their correct worksheet without much extra coding required. There are many other ways that you can accomplish this task, but this is straightforward and simple. If you have a great deal of data, you might want to investigate a more efficient method than creating an additional variable in a data set. Lines 1–5 create the format using the PICTURE statement in PROC FORMAT and date directives discussed in section 2.6, and lines 6–13 create our duplicate variable with our custom format and also subset the data set. Note the date literals used as values in the BETWEEN operator in line 10. We remove the default BY line from the output in line 14. The

SHEET_NAME option allows us to use the BY value token (#BYVAL1) as the name of the worksheet.

```
1    PROC FORMAT;
2    PICTURE namefmt
3    LOW-HIGH='%B %Y' (DATATYPE=date)
4    ;
5    RUN;

6    PROC SQL;
7    CREATE TABLE subset AS
8    SELECT date as sheet_date FORMAT=namefmt15., *
9    FROM sashelp.citiday
10   WHERE date BETWEEN '01JAN1988'd AND '31AUG1988'd
11   ORDER BY DATE
12   ;;;;
13   QUIT;

14   OPTIONS NOBYLINE;
15   ODS TAGSETS.EXCELXP FILE="ex6.1.4.xml" OPTIONS
     (SHEET_NAME="#BYVAL1");
16   PROC PRINT DATA=subset LABEL NOOBS;
17   BY sheet_date;
18   RUN;
19   ODS TAGSETS.EXCELXP CLOSE;
```

This is the resulting workbook:

	A	B	
1	Date of Observation	STOCK MKT INDEX:NY DOW JONES COMPOSITE,	STOC
2	01JAN1988	.	
3	04JAN1988	740.2	
4	05JAN1988	747.38	
5	06JAN1988	750.4	
6	07JAN1988	757.04	
7	08JAN1988	711.36	
8	11JAN1988	721.92	
9	12JAN1988	715.2	
10	13JAN1988	713.59	

January 1988 / February 1988 / March 1988 / April 1988 / May 1988 / June 1988 / July 1988 / August 1988

6.1.4 Using Dates in Macros

While SAS can handle dates, times, and datetime values as numbers, getting the macro processor to accept them as numbers might involve an extra step or two. Let's look at the following code:

```
1   %MACRO date_loop;
2   %DO I='05JUN2014'd %TO '12JUN2014'd;
3      TITLE "Happy Kids Ice Cream Co. Sales for &i";
```

```
4      PROC PRINT DATA =sample.dailysales;
5         ID district;
6         VAR sales;
7         WHERE date EQ &i;
8      RUN;
9   %END;
10  %MEND;
```

We have used two date literals to provide the period over which we want the report run and used it in the title as well. What happens when this is submitted?

```
ERROR: A character operand was found in the %EVAL function or %IF
condition where a numeric operand is required. The condition was:
'05JUN2014'd

ERROR: The %FROM value of the %DO I loop is invalid.

ERROR: A character operand was found in the %EVAL function or %IF
condition where a numeric operand is required. The condition was:
'12JUN2014'd

ERROR: The %TO value of the %DO I loop is invalid.

ERROR: The macro DATE_LOOP will stop executing.
```

To the macro processor, the date literals are only text. They do not function as they do in the rest of SAS. How can we get around this? By using the %SYSEVALF() macro function to force the interpretation of the date literals. Let's replace line 2 with the following line:

```
%DO i=%SYSEVALF('05JUN2014'd) %TO %SYSEVALF('12JUN2014'd);
```

Now the loop will execute properly. However, the date in the title does not look good at all.

1

Happy Kids Ice Cream Co. Sales for 19879

district	sales
Atlanta	$6,000
Kansas City	$75,000
Springfield	$73,000
St. Louis	$15,000

The date in the title doesn't look like a date. We need to make one more modification to the program to get our title to look right. The %LET statement added in line 3 creates a formatted value of the date in the macro loop for use. The PUTN() function is used here instead of the PUT() function because you can't use the PUT() function with %SYSFUNC() or QSYSFUNC().

```
1   %MACRO date_loop;
2   %DO I=%SYSEVALF('05JUN2014'd) %TO %SYSEVALF('12JUN2014'd);
3      %LET TITLEDATE=%SYSFUNC(PUTN(&i,WORDDATE.));
4      TITLE "Happy Kids Ice Cream Co. Sales for &titledate";
5      PROC PRINT DATA=sample.dailysales;
6         ID district;
7         VAR sales;
8         WHERE date EQ &i;
9      RUN;
10  %END;
11  %MEND;
```

And now we get the desired result.

1

Happy Kids Ice Cream Co. Sales for Thursday, June 5, 2014

district	sales
Atlanta	$6,000
Kansas City	$75,000
Springfield	$73,000
St. Louis	$15,000

Our final example of using dates in macros demonstrates how to get a date from a data set into a macro. For this example, we want to put the latest date from a given column into a macro variable so that it can be used in a title. Let's look at some timing variables from this sample data set. We want to put the latest date and time (which is in record 4) from the variable MODDATE (nicely formatted, of course) on the title of each page.

VIEWTABLE: Sample.Pmdata

	crdate	moddate
1	29OCT08:20:27:05	29OCT10:20:27:05
2	13AUG07:14:59:38	13AUG07:14:59:38
3	31OCT10:13:02:05	31OCT10:13:02:05
4	27OCT13:14:28:00	07FEB14:16:18:00
5	04FEB08:14:22:43	01JUN10:09:37:24
6	30MAR13:14:32:49	30MAR13:14:32:49
7	01JUN10:16:04:00	01JUN12:16:04:00
8	23SEP07:10:55:20	01JUN09:09:37:44
9	21JUL02:12:55:45	21JUL02:12:55:45
10	23JUL04:10:52:38	23JUL04:10:52:38
11	03DEC05:17:57:02	03DEC06:17:57:02

Method 1: Using CALL SYMPUT

The CALL SYMPUT() and SYMGET() functions are also used to communicate between the DATA step and the macro world. CALL SYMPUT()takes a value and stores it in macro space from within a DATA step, while SYMGET() takes a macro variable and enables you to use it in a DATA step. In this situation, I recommend that you use CALL SYMPUTX() instead of CALL SYMPUT(), because CALL SYMPUTX() suppresses any trailing blanks in the macro variable created, and the presence of trailing blanks might affect the justification of your title or footnote. The difference between run-time and compile time is very important. You cannot use a %LET statement to store macro values that are defined during execution of a DATA step. You also cannot use a macro variable reference (with an ampersand [&]) in the same DATA step where the macro variable is created with CALL SYMPUT(). The next example uses what was discussed in section 6.1.2 to show how you can obtain a date value from a SAS data set and then put it in a macro variable to use in a title. Since the DATETIME. format isn't formal enough for this report, we're going to create a RPTDATE. custom format.

```
PROC FORMAT;
PICTURE rptdate (DEFAULT=32)
. - .Z = 'Missing'
LOW-HIGH = '%B %d, %Y at %I:%0M %p' (DATATYPE=DATETIME);
RUN;

/* Get the maximum date in the variable */
PROC MEANS DATA=sample.pmdata MAX NOPRINT;
VAR moddate;
OUTPUT OUT=tmp MAX=max;
```

```
RUN;

/*Transfer it to a macro variable using CALL SYMPUTX() */
DATA _NULL_;
SET tmp;
CALL SYMPUTX('lastmod',STRIP(PUT(max,rptdate.)));
RUN;

%PUT lastmod=&lastmod;
```

Partial Log

```
42   %PUT lastmod=&lastmod;
lastmod=February 7, 2014 at 4:18 PM
```

Method 2: Using PROC SQL

For something as simple as the highest (or lowest) value in a date or datetime variable, you don't have to use a SAS procedure and a DATA step. PROC SQL can take the place of both.

```
PROC SQL NOPRINT;
SELECT DISTINCT STRIP(PUT(MAX(moddate),rptdate.)) LENGTH=32 INTO
:lastmod
FROM sample.pmdata;
QUIT;

%PUT lastmod=&lastmod;
```

The Log

```
1   PROC SQL NOPRINT;
2   SELECT DISTINCT STRIP(PUT(MAX(moddate),rptdate.)) LENGTH=32 INTO
3 ! :lastmod
4   FROM sample.pmdata;
5   QUIT;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.08 seconds
      cpu time            0.00 seconds


6
7   %PUT lastmod=&lastmod;
lastmod=February 7, 2014 at 4:18 PM
```

Whichever method you choose, the macro variable &LASTMOD, containing the value "February 7, 2014 at 4:18 PM", is ready to use in your report title.

6.2 Graphing Dates

When you use dates, times, or datetime values in your SAS graphics, you have to remember that they are numbers. This has a large impact on the axes and labeling of your graphics. Attempting to graph a result over a period of time without using formats or intervals will usually result in a graph that is not clear or well-defined. The good news is that you can use all the features associated with SAS dates to improve your graphics, such as formats and intervals. The following series of examples will illustrate. Example 6.5 will demonstrate the graphing of dates with SAS/GRAPH, while Example 6.6 will use ODS graphics to produce a series plot.

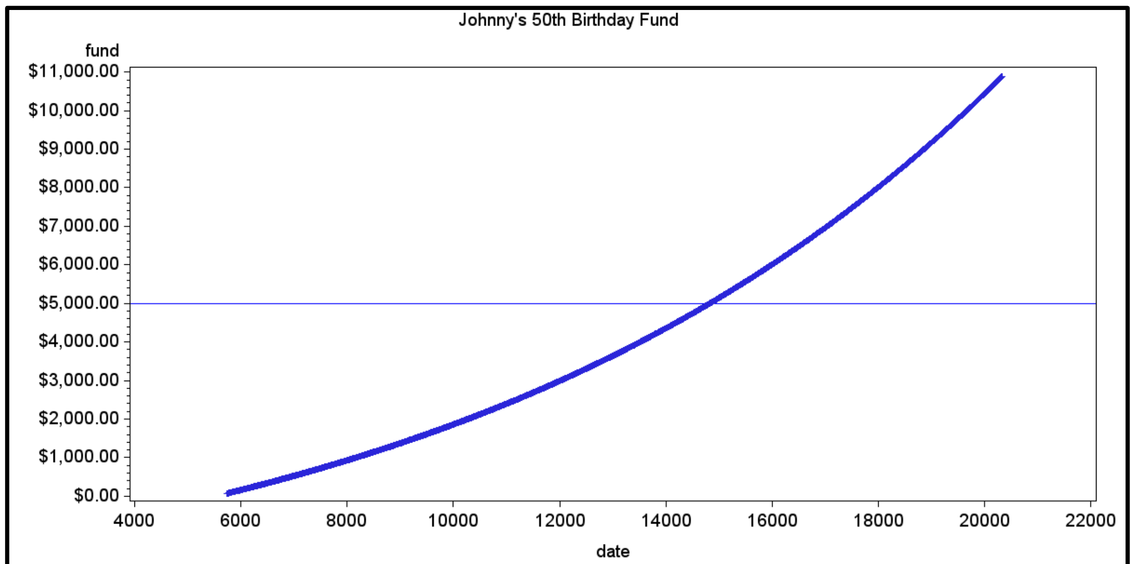
Johnny's Savings Account

When Johnny turned 10 years old on September 1, 1975, he took all the money that he had in his piggy bank and deposited it into a bank account that paid 3.5% interest annually, compounded daily. He promised himself that he would add two dollars at the end of each week and that he would take the money out when he reached the ripe old age of 50. When he was 10, Johnny thought he might have $4,000 by the time he reached age 50. He kept that promise, so let's look at Johnny's earnings from the time he was 10 until he was 50 using the following program:

Example 6.5: SAS/GRAPH

```
TITLE "Johnny's 50th Birthday Fund";
PROC GPLOT DATA=book.graph1;
PLOT fund*date / VREF=5000 LV=1 CV=blue;
RUN;
```

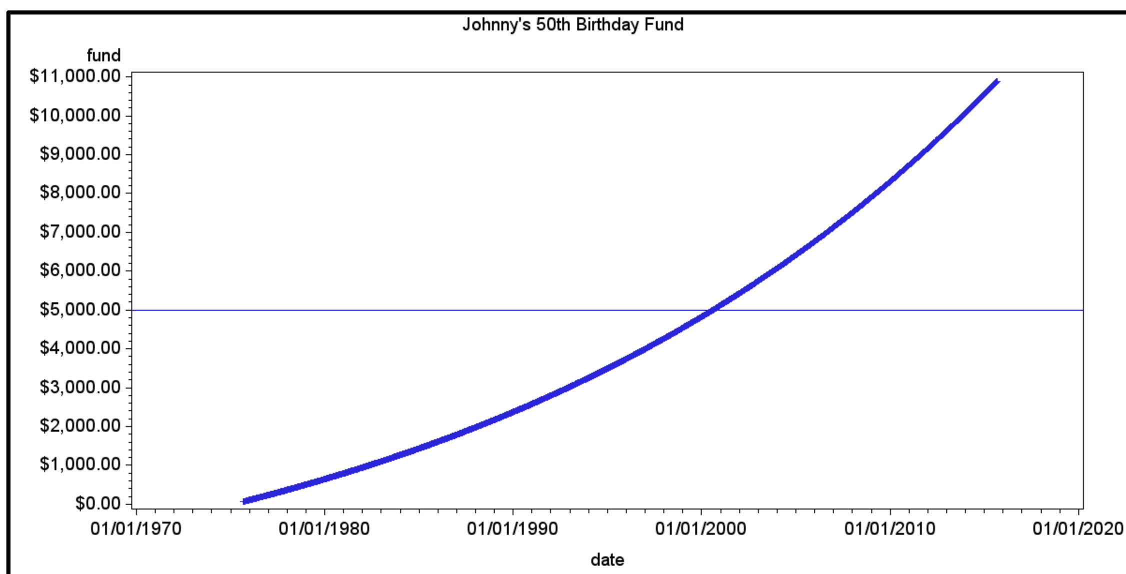
Here is the resulting graph:



Unfortunately, the result of this simple example isn't helpful. We can see that Johnny started around 6000, and he turned 50 somewhere around 20000. What does that mean? This should be easy enough to fix. Don't we just need to add a FORMAT statement?

```
TITLE "Johnny's 50th Birthday Fund";
PROC GPLOT DATA=book.graph1;
PLOT fund*date / VREF=5000 LV=1 CV=blue;
FORMAT date mmddyy10.; /* Add FORMAT statement */
RUN;
```

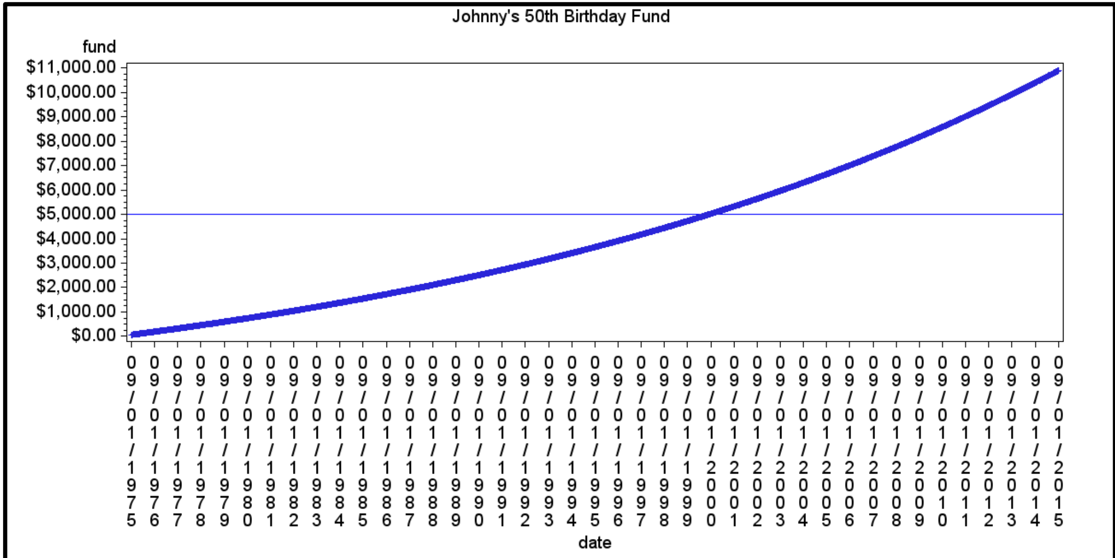
Here is the new GPLOT graph with the FORMAT statement:



What has happened here is that SAS chose the boundaries and figured out major and minor tick marks. In this example, it has selected major intervals at decade boundaries, and minor ones at year boundaries. That's not a bad choice for this example, but SAS/GRAPH doesn't always make such a good choice when selecting the boundaries of an axis. Can't you define the horizontal axis yourself? Of course, you can! Johnny was born in September, so it would make sense to chart his progress at his birthdays and restrict the span of the horizontal axis to the period during which he's contributing.

```
TITLE "Johnny's 50th Birthday Fund";
PROC GPLOT DATA=book.graph1;
PLOT fund*date / VREF=5000 LV=1 CV=blue
                 HAXIS='01SEP1975'd TO '01SEP2015'd by YEAR;
FORMAT date mmddyy10.;
RUN;
```

Here is the new graph that we get from using the HAXIS option in the PLOT statement to specify the axis range:

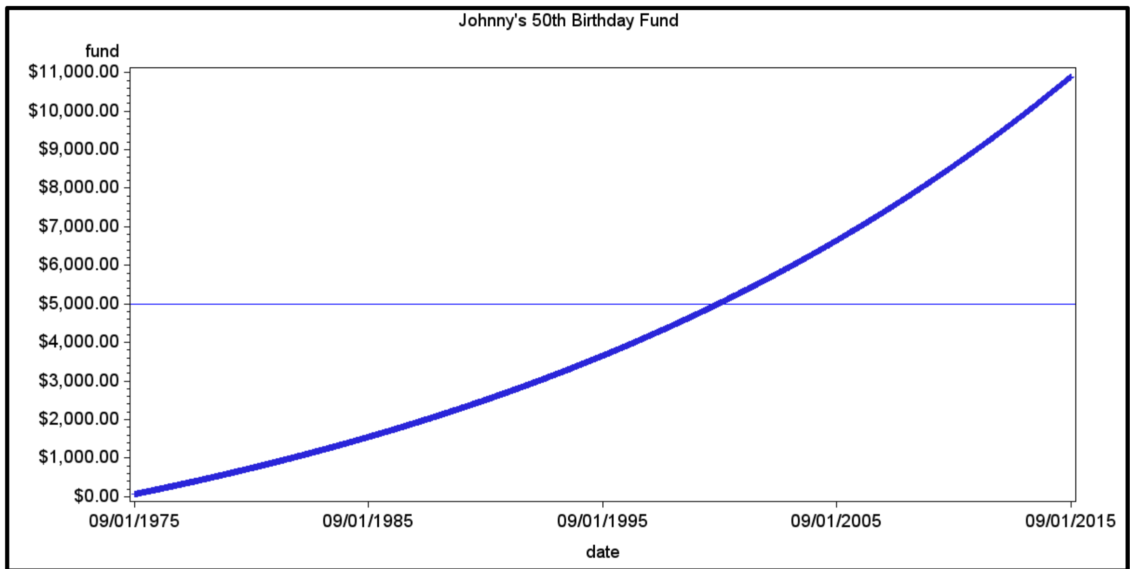


What happened? We defined the horizontal axis as having tick marks *every* year, so SAS accommodated our request. Since there was not enough room to display the values horizontally, SAS automatically rotated the values 90 degrees. Unfortunately, that left less space for the graph itself. We need better spacing on our horizontal axis.

Since decades seemed to work well, let's use those as our intervals, but we want to start on Johnny's 10th birthday, and define the major horizontal axis points at September 1, 1975, September 1, 1985, September 1, 1995, September 1, 2005, and September 1, 2015. An interval multiplier of 10 will create the decade interval, and a shift operator of 69 (60 [months in 5 years] plus 9 [months from January to September]) will move the starting date of the 10-year interval to September 1975 so that it matches the starting point of the horizontal axis. Note that the only change from the previous version is in the interval definition.

```
TITLE "Johnny's 50th Birthday Fund";
PROC GPLOT DATA=book.graph1;
PLOT fund*date / VREF=5000 LV=1 CV=blue
                 HAXIS='01SEP1975'd TO '01SEP2015'd by YEAR10.69;
FORMAT date mmddyy10.;
RUN;
```

Here is the final product:

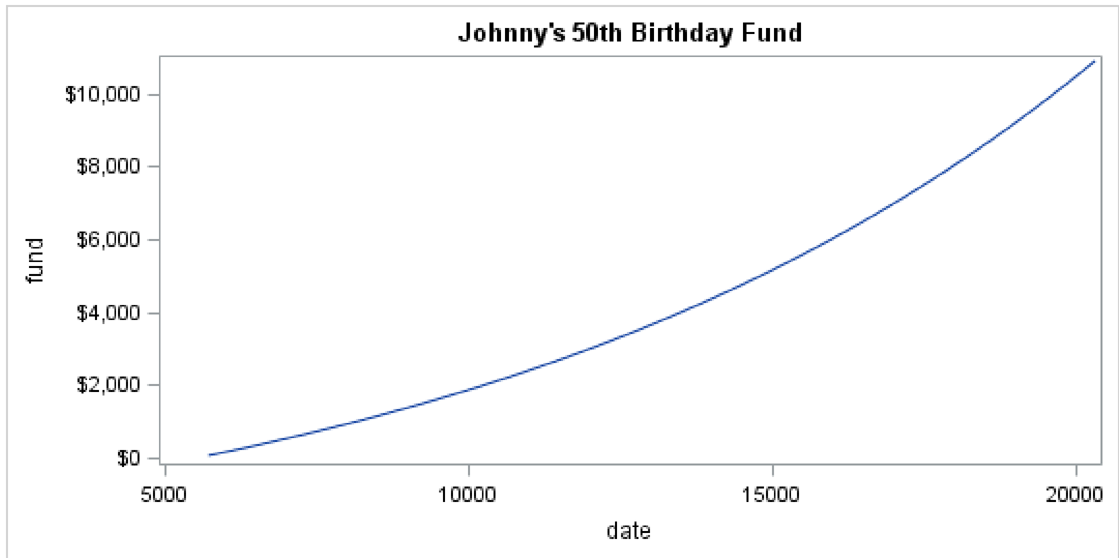


Now that's what we wanted. This demonstrates that you have all of the interval types, as well as their multipliers and shift operators, available when you are defining axes that involve date, time, and datetime values in SAS/GRAPH. It makes defining the exact scope of the graph much easier, not to mention comprehending what you've graphed.

Example 6.6: ODS Graphics

We will continue to use Johnny's 50th birthday fund for the SGPLOT series of examples. The SGPLOT procedure has a way of assigning a time scale to an axis by setting the axis TYPE option in the XAXIS statement equal to TIME, and the SERIES style of plot is perfect for graphs of longitudinal data.

Here's the basic SGPLOT from the above code.



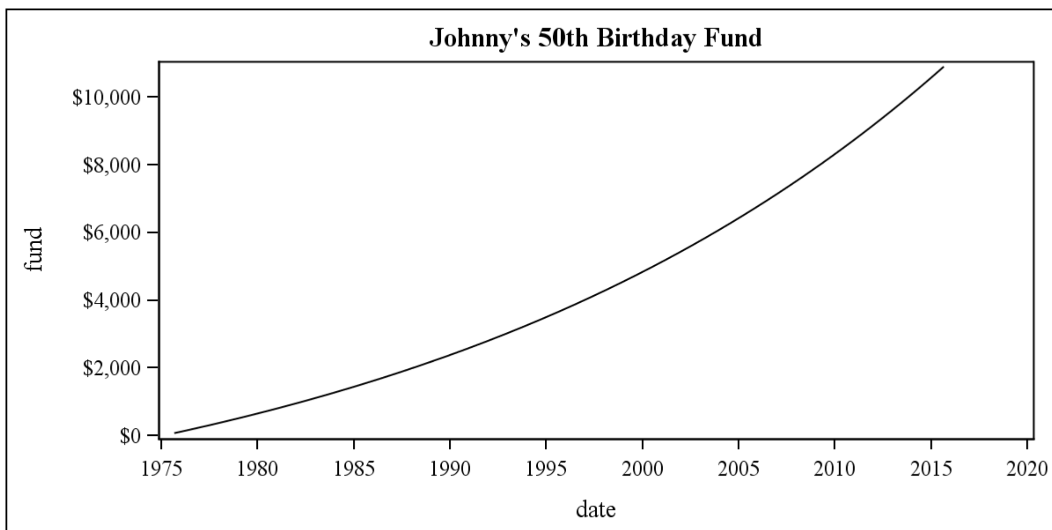
As with the GPLOT example, an unformatted date variable isn't very useful. However, even though we got our graph, SGPLOT did not run smoothly. In order to produce this graph, the procedure made some adjustments in order to produce the graph. Let's look at the log from the program that produced the above graph.

```
3     TITLE "Johnny's 50th Birthday Fund";
4     PROC SGPLOT DATA=book.graph1;
5     XAXIS TYPE=time;
6     SERIES X=date Y=fund;
7     RUN;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           1.20 seconds
      cpu time            0.09 seconds

NOTE: Time axis can only support date time values. The axis type
will be changed to LINEAR.    ❶
NOTE: There were 2435 observations read from the data set
      BOOK.GRAPH1.
```

Note ❶ is telling us that you can't use the TYPE value of TIME. But aren't we graphing dates? The SGPLOT procedure relies on formats to determine that the data being graphed are dates or times. Without it, SGPLOT errs on the side of caution and just assumes that you are graphing linear data. This will create a plot for data graphed along a sequence (such as dates), so it looks right, but it is not clear that you are graphing dates.

Let's add the missing format to our SGPLOT:

```
TITLE "Johnny's 50th Birthday Fund";
PROC SGPLOT DATA=book.graph1;
XAXIS TYPE=time;
SERIES X=date Y=fund;
FORMAT date monyy7.;
RUN;
```



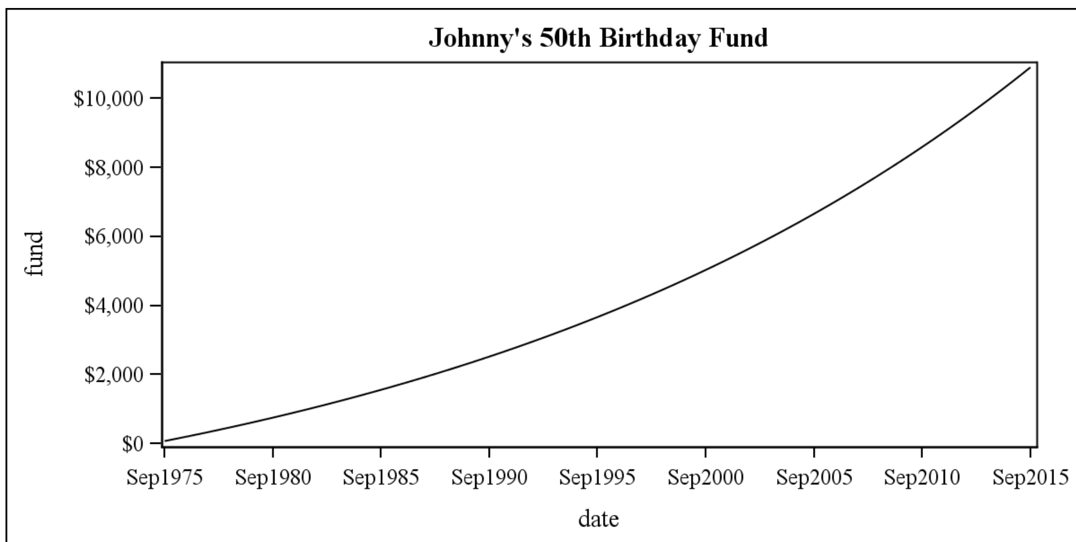
That looks better, even though SAS did not use the MONYY7. format that you requested and substituted one of its own as seen in note ❶ below.

```
51   TITLE "Johnny's 50th Birthday Fund";
52   PROC SGPLOT DATA=book.graph1;
53   XAXIS TYPE=time;
54   SERIES X=date Y=fund;
55   FORMAT date monyy7.;
56   RUN;

NOTE: PROCEDURE SGPLOT used (Total process time):
      real time           0.90 seconds
      cpu time            0.20 seconds

❶ NOTE: The column format MONYY7 is replaced by an auto-generated
format
      on the axis.
```

This is really the same problem as with SAS/GRAPH when you haven't defined the tick marks on the X axis properly. The VALUES= option in the XAXIS statement allows us to define our scale in the same way that the HAXIS= option in the PLOT statement does in SAS/GRAPH. For a 50-year period, let's try 5-year intervals, starting when Johnny made his promise on September 1, 1975.

```
TITLE "Johnny's 50th Birthday Fund";
PROC SGPLOT DATA=book.graph1;
XAXIS TYPE=time VALUES=('01sep1975'd TO '01sep2015'd BY YEAR5);
SERIES X=date Y=fund;
FORMAT date monyy7.;
RUN;
```

Now the display looks much nicer, and we have our MONYY7. format displayed.



6.3 The Basics of PROC EXPAND

The EXPAND procedure is a part of SAS/ETS, which is used to prepare time series data for further analysis. It creates a SAS data set and does not produce printed output.

6.3.1 Capabilities of PROC EXPAND

PROC EXPAND will change the sampling frequency of the data that you have and convert it to a different one. It can interpolate values in time series data, for example, when you have quarterly data that you need to report or analyze on a monthly basis. It can perform the reverse operation, that is, to aggregate (collapse) data from a higher sampling frequency to a lower one, such as taking monthly data and turning it into quarterly data. PROC EXPAND can interpolate missing values

even if you aren't changing the sampling frequency. It also provides for extensive data transformations and performs all of these functions without a lot of DATA step programming. The SAS/ETS documentation provides detail on the procedure, its statements, and the options for those statements. There are other procedures available in SAS/ETS, such as TIMESERIES and TIMEDATA, that can be used to manipulate time series data, and if you need to work with time series data on a regular basis, you should investigate the SAS/ETS product and its documentation more carefully.

PROC EXPAND uses SAS interval definitions. This includes interval multipliers and the shift index. For a detailed explanation of intervals, interval multipliers, and the interval shift index, see Sections 5.5 and 5.6 of this book. When you use these interval definitions (plus any shift index and/or multipliers), PROC EXPAND will automatically adjust for any calendar effects (leap years, varying number of days in a month). As with anything that uses these interval definitions, all measurements and calculations are considered to be at the beginning of the interval(s) specified. It is possible to change that definition with options in one of the PROC EXPAND statements, and those are discussed in Section 6.3.5.

PROC EXPAND Sample Data

The following data set will be used for the examples in this section. This is monthly total light rail ridership data obtained from the American Public Transportation Association for the years 2003 and 2004 and is used with their permission. The values for October and November 2003 were removed from the data to demonstrate some of the capabilities of PROC EXPAND.

Date	Riders (thousands)
JAN2003	2679.9
FEB2003	2421.9
MAR2003	2704.6
APR2003	2778.3
MAY2003	2718.6
JUN2003	2618.2
JUL2003	2999.0
AUG2003	3504.7
SEP2003	3329.4
OCT2003	.
NOV2003	.
DEC2003	2888.6
JAN2004	3132.9
FEB2004	2814.3
MAR2004	3067.3
APR2004	2928.8

Date	Riders (thousands)
MAY2004	2958.3
JUN2004	2966.3
JUL2004	3000.8
AUG2004	3071.2
SEP2004	2958.9
OCT2004	2992.8
NOV2004	3017.5
DEC2004	3038.4

6.3.2 Using PROC EXPAND to Convert to a Higher Frequency

You can use PROC EXPAND to convert data from a lower sampling frequency to a higher sampling frequency (for example, converting monthly data to daily or weekly data). It does so by interpolation, and the syntax to convert to a higher sampling frequency is as follows:

Example 6.7: Converting to a Higher Sampling Frequency

```
1   PROC EXPAND DATA=book.month OUT=seven_days FROM=MONTH TO=WEEK;
2   ID date;
3   CONVERT riders;
4   RUN;
```

The PROC EXPAND statement in line 1 specifies the output data set (SEVEN_DAYS) and explains how the data in BOOK.MONTH should be converted, from MONTH intervals to WEEK intervals. The ID statement in line 2 indicates the variable that identifies the time of each record. Since the WEEK interval begins on Sunday, the dates will be aligned on Sundays. If you want the dates to align to a different day of the week, use a shift indicator. The WEEK.2 interval will align the dates to Mondays; the WEEK.3 interval aligns to Tuesdays, and so on.

You will usually use an ID statement with PROC EXPAND. Otherwise, SAS will create an ID variable for the input records and use the starting point of January 1, 1960, which might not be what you want. The CONVERT statement identifies the variable(s) to convert. You can also rename the variable(s) being converted in the output data set like this: *CONVERT input-*

var=output-var; Here are the first eight observations from the data set SEVEN_DAYS produced by the above code:

date	Riders (thousands)
29DEC2002	2770.23
05JAN2003	2573.57
12JAN2003	2449.76
19JAN2003	2393.52
26JAN2003	2391.24
02FEB2003	2429.29
09FEB2003	2494.03
16FEB2003	2571.86

When you interpolate time series data to a higher frequency, consider that the interpolated values do not add any new information to the data because they have been derived mathematically, not observed. Therefore, any statistical analyses performed with these interpolated data should be viewed with caution.

6.3.3 Using PROC EXPAND to Convert to a Lower Frequency

You can convert data to a lower frequency with PROC EXPAND in two ways: First, you can use the same syntax as with converting to a higher sampling frequency, except that the TO= interval would be of a lower sampling frequency. When you convert your data this way, PROC EXPAND performs interpolation for missing values using a curve-fitting method, which allows for conversion between intervals that aren't nested. A nested interval is one that fits wholly inside another interval (for example, days nest within weeks, because there are exactly seven days in a week, but weeks do not nest within months, because most months have partial weeks). The following program will interpolate any missing values in our data by fitting a cubic spline function through the existing data before converting the data from a month frequency to a quarter frequency.

Example 6.8: Converting to a Lower Frequency

```
PROC EXPAND DATA=book.month OUT=quarterly FROM=MONTH TO=QTR;
ID date;
CONVERT riders;
RUN;
```

Obs	date	Riders (thousands) After Interpolation
1	01JAN2003	2679.90
2	01APR2003	2778.30

Original Values From BOOK.MONTH
2679.9
2778.3

Obs	date	Riders (thousands) After Interpolation
3	01JUL2003	2999.00
4	01OCT2003	2993.28
5	01JAN2004	3132.90
6	01APR2004	2928.80
7	01JUL2004	3000.80
8	01OCT2004	2992.80

Original Values From BOOK.MONTH
2999.0
.
3132.9
2928.8
3000.8
2992.8

The resulting data set QUARTERLY (above) has eight observations, four for each year, synchronized on the QTR interval boundaries. As you can see, the data that were missing from our original data set (for October 2003) are interpolated.

The second method enables you to perform simple aggregation (addition) without interpolation of missing values. The AGGREGATE method always produces an exact result without interpolation, and it requires that the intervals be nested. This program shows the result of a simple aggregation on our sample data:

Example 6.9: Performing Aggregation without Interpolation

```
PROC EXPAND DATA=book.month OUT=annual FROM=MONTH TO=YEAR;
ID date;
CONVERT riders / METHOD=AGGREGATE;
RUN;
```

date	Riders (thousands)	
2003	.	There are 2 missing observations for this year, which yields a missing result.
2004	35947.5	Total across all 12 months.

Example 6.10: The Importance of the ID Statement in PROC EXPAND

The following PROC EXPAND step has no ID statement. Let's run it so that we can see the assumptions that SAS makes in its absence. This illustrates why the ID statement is almost always used with this procedure:

```
PROC EXPAND DATA=book.month OUT=ANNUAL FROM=MONTH TO=YEAR;
CONVERT riders;
RUN;
```

date	**Riders (thousands)**
01JAN1960	2679.9
01JAN1961	3132.9

How did we wind up with data for 1960 and 1961 when we used data from 2003 and 2004? If there is no ID variable to indicate the dates, SAS will create ID values to label the input records, and it will start from its zero point, January 1, 1960.

6.3.4 Using PROC EXPAND to Interpolate Missing Values

PROC EXPAND can also be used to interpolate missing values without converting frequencies. There are two ways to do this; use the one that fits your situation. If you are interpolating missing values at specific points in time, leave off the FROM= and TO= options, but make sure that you use an ID statement to indicate the variable that contains the time points of the observed values. The time points do not have to be evenly spaced, nor do you need a record for each time point within the interval. PROC EXPAND will read the values supplied in the ID variable and will fit a spline function through the available data points. It then fills in the missing values based on that fitted spline. Remember that the data for the months of October and November are missing in our sample table. The following program demonstrates:

Example 6.11: Interpolating Missing Values

```
PROC EXPAND DATA=book.month OUT=nomiss;
ID date;
CONVERT riders;
RUN;
```

The bolded and italicized values in the following table are the result of the interpolation performed by PROC EXPAND.

date	**Riders (thousands)**
01JAN2003	2679.90
01FEB2003	2421.90
01MAR2003	2704.60
01APR2003	2778.30
01MAY2003	2718.60
01JUN2003	2618.20
01JUL2003	2999.00
01AUG2003	3504.70

date	Riders (thousands)
01SEP2003	3329.40
01OCT2003	***2993.28***
01NOV2003	***2788.68***
01DEC2003	2888.60

The second method interpolates missing values in a time series without converting the observation frequency. Use this when you want to fill in the missing values and maintain the same observation frequency. It requires the FROM= option, but leave off the TO= option, as shown here:

```
PROC EXPAND DATA=book.month OUT=nomiss2 FROM=MONTH;
ID date;
CONVERT riders;
RUN;
```

By default, the interpolation is performed by fitting the points to a cubic spline curve. You can request other methods of interpolation with the METHOD= option in the CONVERT statement, and these are detailed in the SAS/ETS documentation. PROC EXPAND will ignore observations that have missing values for the ID variable, even if there are data points for the CONVERT variable(s). Table 6.1 is a summary of what PROC EXPAND does when there are missing values for the ID variable and/or CONVERT data points.

Table 6.1: How PROC EXPAND Handles Interpolation of Missing Values in Input Data

ID Variable	Data	PROC EXPAND Will
Missing	*Missing*	Interpolate
Not Missing	*Missing*	Interpolate
Missing	Not Missing	Ignore

6.3.5 The OBSERVED= Option for the CONVERT Statement in PROC EXPAND

As with the other uses of SAS date, time, and datetime intervals, the default for PROC EXPAND is to consider the values as being from the beginning of the intervals provided in the FROM= and TO= options. This is not always the case with real-world data, and it can cause very different results, especially if the values are not measured at the beginning of the given interval(s), or they do not represent a single observed value for a specific point in time. You can control how the SAS

intervals are used with the OBSERVED option in the CONVERT statement. There are six possible values for the OBSERVED= option, as shown in Table 6.2:

Table 6.2: Values for the OBSERVED= Option

Values	Description
BEGINNING	Beginning of the period
MIDDLE	Middle of the period
END	End of the period
TOTAL	Totals for the period
AVERAGE	Averages across the period
DERIVATIVE	Only valid as the 'to' value when the cubic spline function is the conversion method

The syntax of the OBSERVED= option is:

1. OBSERVED=*value* OR
2. OBSERVED=(*from-value*, *to-value*)

value is one of the keywords in table 6.2, *from* indicates the characteristics of the data from which you are converting, and *to* represents the characteristics of the data resulting from the conversion. Using form 1 is the same as specifying the same value for both *from* and *to*; that is, OBSERVED=AVERAGE is the same as OBSERVED=(AVERAGE,AVERAGE). Again, if you do not supply an OBSERVED option for the conversion, it is the same as using OBSERVED=(BEGINNING,BEGINNING).

Examples 6.12 and 6.13 demonstrate how different combinations of the OBSERVED= option affect our sample data when we increase and decrease the sampling frequency. The program below shows the code to increase the sampling frequency. It also shows how to rename the output variables in the CONVERT statement by placing an equal sign (=) after the data set variable and providing the new variable name afterward.

Example 6.12: Effect of Different Values for OBSERVED= Option on Increased Frequency

```
/* The effect of the different values for the OBSERVED option of
   the CONVERT statement in PROC EXPAND on increased sample
frequency  */

PROC EXPAND DATA=book.month OUT=seven1 FROM=MONTH TO=WEEK;
ID date;
CONVERT riders=beginning / OBSERVED=BEGINNING /*Default */;
RUN;
PROC EXPAND DATA=book.month OUT=seven2 FROM=MONTH TO=WEEK;
```

```
ID date;
CONVERT riders=middle / OBSERVED=MIDDLE;
RUN;
PROC EXPAND DATA=book.month OUT=seven3 FROM=MONTH TO=WEEK;
ID date;
CONVERT riders=end / OBSERVED=END;
RUN;
PROC EXPAND DATA=book.month OUT=seven4 FROM=MONTH TO=WEEK;
ID date;
CONVERT riders=total / OBSERVED=TOTAL;
RUN;
PROC EXPAND DATA=book.month OUT=seven5 FROM=MONTH TO=WEEK;
ID date;
CONVERT riders=average / OBSERVED=AVERAGE;
RUN;
PROC EXPAND DATA=book.month OUT=seven6 FROM=MONTH TO=WEEK;
ID date;
CONVERT riders=begend / OBSERVED=(BEGINNING,END);
RUN;
PROC EXPAND DATA=book.month OUT=seven7 FROM=MONTH TO=WEEK;
ID date;
CONVERT riders=avetot / OBSERVED=(AVERAGE,TOTAL);
RUN;

DATA compare_lo;
MERGE seven1 seven2 seven3 seven4 seven5 seven6 seven7;
BY date;
LABEL
   beginning="BEGINNING" middle = "MIDDLE" end = "END" average =
"AVERAGE"
   total = "TOTAL" begend = "BEGINNING,END" avetot = "AVERAGE,TOTAL"
;;;
FORMAT beginning--avetot 7.2;
RUN;

PROC REPORT DATA=compare_lo NOWD SPLIT='\';
COLUMNS date ('OBSERVED=\Option Value' beginning middle end average
total);
FORMAT date DATE9.;
DEFINE date / display;
WHERE DATE LT '01MAR2003'd;
RUN;

PROC REPORT DATA=compare_lo NOWD SPLIT='\';
COLUMNS date ('OBSERVED=\Option Value' begend avetot);
FORMAT date DATE9.;
DEFINE date / display;
WHERE DATE LT '01MAR2003'd;
RUN;
```

OBSERVED= Option Is the Same for FROM= and TO=:

	OBSERVED= Option Value				
date	**BEGINNING**	**MIDDLE**	**END**	**AVERAGE**	**TOTAL**
29DEC2002	2770.19	.	.	3260.80	580.48
05JAN2003	2573.61	.	.	2897.25	597.97
12JAN2003	2449.83	2708.88	.	2638.87	608.44
19JAN2003	2393.59	2536.09	.	2472.89	613.17
26JAN2003	2391.28	2436.82	2652.65	2384.91	613.55
02FEB2003	2429.28	2398.83	2506.22	2360.52	610.98
09FEB2003	2493.99	2409.06	2428.26	2385.33	606.85
16FEB2003	2571.80	2454.40	2406.10	2444.93	602.56
23FEB2003	2649.08	2521.78	2427.05	2524.93	599.50

As you can see, the OBSERVED= option has a very large effect on the results that PROC EXPAND yields. The BEGINNING column is the default, and that is the interpolation calculated if the numbers are measured at the beginning of the month and converted to the BEGINNING of the week. MIDDLE and END do the calculation as if the numbers were measured and converted at the middle and the end of the month, respectively. That is why the interpolated values are missing in those columns in the above chart. The numbers are not available until the beginning of the interval (the beginning of the week containing the middle and end, respectively, of the month).

If you are measuring totals (which we are, since this is mass transit ridership data), the values are radically different. TOTAL means that the number being interpolated is not representative of a single point in the FROM= interval, but that it is obtained across the duration of the FROM= interval. Therefore, the numbers in that column of the above table represent the number of riders per week, since that is the interval to which we are converting our data. AVERAGE considers the numbers to be averages for both the FROM and TO= intervals.

It gets a little more interesting when we use different values for the OBSERVED option, but it is extremely important to remember that these are mathematically derived and not data-based, so you need to exercise some caution and judgment as to the usefulness of these numbers. The BEGINNING,END combination takes the original value as the value at the beginning of the MONTH interval, and then (using, in this case, the default SPLINE method) returns the value from that curve at the end of the WEEK interval. Similarly, the AVERAGE,TOTAL combination considers the original value to be the monthly average and calculates a weekly total based on the default SPLINE method. There are other methods available to use in the conversion of intervals, and I encourage you to read the SAS/ETS documentation for more in-depth information about converting to a higher sampling frequency.

Different FROM and TO Values in the OBSERVED= Option

	OBSERVED= Option Value	
date	**BEGINNING,END**	**AVERAGE,TOTAL**
29DEC2002	2573.61	22825.6
05JAN2003	2449.83	20280.8
12JAN2003	2393.59	18472.1
19JAN2003	2391.28	17310.2
26JAN2003	2429.28	16694.3
02FEB2003	2493.99	16523.6
09FEB2003	2571.80	16697.3
16FEB2003	2649.08	17114.5
23FEB2003	2712.25	17674.5

In contrast to Example 6.12, Example 6.13 will show the effect of the OBSERVED= option on a lower sampling frequency. Remember, since our original data had missing values, some interpolation will take place.

Example 6.13: Effect of Different Values for OBSERVED= Option on Lowered Frequency

```
/* The effect of the different values for the OBSERVED option of
   the CONVERT statement in PROC EXPAND with a decreased sample
frequency  */
PROC EXPAND DATA=book.month OUT=annual1 FROM=MONTH TO=YEAR;
ID date;
CONVERT riders=beginning / OBSERVED=BEGINNING /*Default */;
RUN;
PROC EXPAND DATA=book.month OUT=annual2 FROM=MONTH TO=YEAR;
ID date;
CONVERT riders=middle / OBSERVED=MIDDLE;
RUN;
PROC EXPAND DATA=book.month OUT=annual3 FROM=MONTH TO=YEAR;
ID date;
CONVERT riders=end / OBSERVED=END;
RUN;
PROC EXPAND DATA=book.month OUT=annual4 FROM=MONTH TO=YEAR;
ID date;
CONVERT riders=total / OBSERVED=TOTAL;
RUN;
PROC EXPAND DATA=book.month OUT=annual5 FROM=MONTH TO=YEAR;
```

```
ID date;
CONVERT riders=average / OBSERVED=AVERAGE;
RUN;
PROC EXPAND DATA=book.month OUT=annual6 FROM=MONTH TO=YEAR;
ID date;
CONVERT riders=begend / OBSERVED=(BEGINNING,END);
RUN;
PROC EXPAND DATA=book.month OUT=annual7 FROM=MONTH TO=YEAR;
ID date;
CONVERT riders=avetot / OBSERVED=(AVERAGE,TOTAL);
RUN;

DATA compare_hi;
MERGE annual1 annual2 annual3 annual4 annual5 annual6 annual7;
BY date;
LABEL
   beginning="BEGINNING" middle = "MIDDLE" end = "END" average =
"AVERAGE"
   total = "TOTAL" begend = "BEGINNING,END" avetot = "AVERAGE,TOTAL"
;;;
FORMAT beginning--avetot 11.2;
RUN;

PROC REPORT DATA=compare_hi NOWD SPLIT='\';
COLUMNS date ('OBSERVED=\Option Value' beginning middle end average
total);
FORMAT date YEAR4.;
DEFINE date / display;
RUN;

PROC REPORT DATA=compare_hi NOWD SPLIT='\';
COLUMNS date ('OBSERVED=\Option Value' begend avetot);
FORMAT date YEAR4.;
DEFINE date / display;
RUN;
```

OBSERVED= Option Is the Same for FROM= and TO=:

	OBSERVED= Option Value				
date	**BEGINNING**	**MIDDLE**	**END**	**AVERAGE**	**TOTAL**
2003	2679.90	2764.22	2888.60	2861.76	34476.22
2004	3132.90	2972.48	3038.40	2996.92	35947.50

BEGINNING, MIDDLE, and END don't give us a very good idea of yearly ridership, because they are considering the entire ridership as occurring on the beginning, middle, or end of each observation in the FROM= interval. TOTAL is the value for the entire year, and AVERAGE is

calculated on the monthly values. However, all of these values are calculated with interpolation of the missing values in October and November 2003. Below, the interpolation for those missing values is still performed even though we are using different values for the *from* and *to* observation characteristics. Of special interest is the AVERAGE,TOTAL column. It considers the monthly data to be the average monthly ridership, so PROC EXPAND interpolates for the missing data and then provides a ridership total for each year based on that average monthly ridership.

Different FROM and TO Values in the OBSERVED= Option

	OBSERVED= Option Value	
date	**BEGINNING,END**	**AVERAGE,TOTAL**
2003	3132.90	1044544.20
2004	3153.51	1096872.40

PROC EXPAND has many more capabilities, and the preceding examples give only the most basic information about how to use this powerful procedure with time series data. You can refer to the documentation for SAS/ETS to get a much more complete explanation of PROC EXPAND and its options.

6.4 International Date, Time, and Datetime Formats and Informats

Version 9 of SAS has formats for dates and times in languages other than U.S. English. It is included in Base SAS as a part of National Language Support (NLS). The key to NLS is in the LOCALE= or DFLANG= system options. The default value for the LOCALE option is defined in the SAS configuration file and is set during installation, but it can be changed with an OPTIONS statement or inside the OPTIONS window. The LOCALE= option implicitly sets two other options that can affect dates, times, and datetime values in SAS. It will set the DATESTYLE= option, which determines how the ANYDT informats will interpret character strings where the order of month, day, and year is ambiguous. The DFLANG= option defines the default language that SAS will use.

There are a few specific date formats for Taiwanese, Japanese, and Hebrew, but you can consider the majority of international formats and informats as falling into one of two informal categories: the "EUR" category or the NLS category. In general, it is recommended that you use the NLS formats and informats because you do not have to write code specific to a language. Changing languages using the NLS facility in SAS is a matter of changing the value of the LOCALE= option, which brings the formats in line with the rest of the SAS session. However, the "EUR" category can be useful. You can select the language based on either the DFLANG= system option or by replacing the "EUR" in the format name with a specific language abbreviation. Using the language abbreviations is handy if you are working with many languages on the same output, because they enable you to specify the language without regard to a system option, and you can use them exactly where you need them, for as long as you need them.

6.4.1 "EUR" Formats and Informats

Each of these formats and informats correspond to an English language format or informat. However, the minimum, maximum, and default widths for the format or informat are dependent upon the language being used at the time. Tables 6.3 and 6.4 list the English language formats and their EUR format names, and the EUR informats.

Table 6.3: International Format Names and Their English Language Equivalents

English Language Format Name	International Format Name
DATE.	EURDFDE.
DATETIME.	EURDFDT.
DDMMYY.	EURDFDD.
DOWNAME.	EURDFDWN.
MONNAME.	EURDFMN.
MONYY.	EURDFMY.
WEEKDATX.	EURDFWKX.
WEEKDAY.	EURDFDN.
WORDDATX.	EURDFWDX.

Table 6.4: International Informat Names and Their English Language Equivalents

Informat	Description
EURDFDEw.	Reads international date values in the form ddmonyy(yy), where *dd* represents the day of the month, *mon* is the three-letter month abbreviation in the language specified by the DFLANG= system option or by the appropriate three-letter prefix, and *yy(yy)* is the two- or four-digit year.
EURDFDTw.	Reads international datetime values in the form ddmonyy hh:mm:ss.ss or ddmonyyyy hh:mm:ss.ss.
EURDFMYw.	Reads month and year date values in the form monyy or monyyyy.

You replace "EUR" with a specific three-letter language prefix in any of the above formats or informats to define the language that you want to use. This overrides the DFLANG= system option and is a good way to display dates in multiple languages simultaneously. Table 6.5 is a list of all the valid languages with their three-letter prefix. In addition, we'll show the effect of using each three-letter prefix on the EURDFWKX. format by using the reference date of Tuesday, February 18, 2014. As a comparison, the table also includes the reference date formatted as the English equivalent using the WEEKDATX. format.

Table 6.5: International Date Formats with Language Abbreviations

Language Prefix	Language	Format Name	Formatted Date
		WEEKDATX.	Tuesday, 18 February 2014
AFR	Afrikaans	AFRDFWKX.	Dinsdag, 18 Februarie 2014
CAT	Catalan	CATDFWKX.	Dimarts, 18 Febrer 2014
CRO	Croatian	CRODFWKX.	utorak, 18 veljača 2014
CSY	Czech	CSYDFWKX.	úterý, 18 únor 2014
DAN	Danish	DANDFWKX.	tirsdag, den 18. februar 2014
NLD	Dutch	NLDDFWKX.	dinsdag, 18 februari 2014
FIN	Finnish	FINDFWKX.	Tiistaina, 18. helmikuuta 2014
FRA	French	FRADFWKX.	Mardi 18 février 2014
DEU	German	DEUDFWKX.	Dienstag, 18. Februar 2014
HUN	Hungarian	HUNDFWKX.	2014.február 18., kedd
ITA	Italian	ITADFWKX.	Martedì, 18 Febbraio 2014
MAC	Macedonian	MACDFWKX.	vtornik, 18 fevruari 2014
NOR	Norwegian	NORDFWKX.	tirsdag, 18. februar 2014
POL	Polish	POLDFWKX.	wtorek, 18 luty 2014
PTG	Portuguese	PTGDFWKX.	Terça-feira, 18 de fevereiro de 2014
RUS	Russian	RUSDFWKX.	Вторник, 18 Февраль 2014
ESP	Spanish	ESPDFWKX.	martes, 18 de febrero de 2014
SLO	Slovenian	SLODFWKX.	torek, 18 februar 2014
SVE	Swedish	SVEDFWKX.	Tisdag, 18 februari 2014
FRS	Swiss_French	FRSDFWKX.	Mardi 18 février 2014
DES	Swiss_German	DESDFWKX.	Dienstag, 18. Februar 2014

6.4.2 NLS Formats

The output from the NLS series of formats is defined by the LOCALE= system option. Unlike the "EUR" series, you cannot specify a language; the language is defined by the current value of the LOCALE= option. Use these formats when your output might be generated in several locations around the world, but you don't have to display multiple languages within the same output. These formats work by converting a SAS date, time, or datetime value to that of the specified locale and then formatting the result. These formats are also noteworthy in that the result is *left-justified*, as opposed to the right-justification of most of the other date, time, and datetime formats. This is true for all ODS destinations as well as for traditional column-based output. Of course, with ODS destinations, the justification of the column will be performed according to any STYLE in effect.

Tables 6.6, 6.7, and 6.8 list the NLS formats available for dates, datetimes, and times, respectively. Each table will give the NLS format name, a description of the output created by the format, the default format width, and width range. The description of the output can also contain a recommended width, which might differ from the default. The recommended width is the minimum format width necessary to display all possible date, datetime, or time values, because the length of the output might exceed the default width in the given format for some locale and encoding combinations. If you use the recommended widths given in the table(s) below, you will always get accurate output. Otherwise, your output could have a series of asterisks (*****) in place of the date string that you expected. In general, it is always better to specify a format width that is too long rather than one that is too short. ODS will handle most justification issues arising from overestimating how many characters will be returned from a format. Note that the width specifications for datetime and time formats can accommodate fractional seconds (*w.d*), and they will be displayed with the locale-specific decimal separator. Because there are so many NLS formats available, Appendix B will show the difference in output resulting from different LOCALE settings using a specific date, datetime, and time as appropriate for each of the NLS formats in the following three tables.

Table 6.6: NLS Date Formats

Format Name	Description	Default Width	Width Range
NLDATE*w*.	Displays the date as month name, day, and year in local format. SAS will use DATE. in local format or abbreviate the month name to fit the format width specified. It is recommended that you use a minimum format width of 25 to ensure accurate output across all supported languages.	20	10–200
NLDATEL*w*.	Displays the date as month name, day, and year in local format. SAS will abbreviate month name or use only numbers and delimiters to fit the format width specified.	18	2–220
NLDATEM*w*.	Displays the date as the local abbreviation for month along with the day and year, but will use only numbers and delimiters to fit the format width specified.	10	2–200
NLDATEMD*w*.	Displays month name and day (no year) from a date value. SAS will use the local abbreviation for the month name if the format width cannot accommodate the full month name.	16	6–200
NLDATEMDL*w*.	Displays the full month name and day (no year) from a date value in local format. SAS will abbreviate the month name or use only numbers and delimiters for the month and day if the format width cannot accommodate the full month name.	12	5–200

Format Name	Description	Default Width	Width Range
NLDATEMDM*w*.	Displays the local abbreviation for the month name and the day (no year) from a date value. SAS will use only numbers and delimiters for the month and day to fit the format width specified.	9	5–200
NLDATEMDS*w*.	Displays the month and day (no year) from a date value using only numbers and delimiters.	5	5–200
NLDATEMN*w*.	Displays the month name from a date value in local format. SAS will abbreviate the month name to fit the format width specified.	9	4–200
NLDATES*w*.	Displays the date in local format using numbers and delimiters only.	10	2–200
NLDATEW*w*.	Displays a date value as day of the week and date in local format. SAS will abbreviate day-of-week, and/or month name as necessary to fit the format width given.	29	10–200
NLDATEWN*w*.	Displays a date value as the day of the week in local format. SAS will abbreviate as necessary to fit the format width given.	9	4–200
NLDATEYM*w*.	Displays month name and year from a date value in local format. SAS will abbreviate month name and/or use 2-digit year as necessary to fit the format width. **Note that some format widths might be too small to accommodate the abbreviations. In that case, a series of asterisks (*****) will be displayed.**	16	6–200
NLDATEYML*w*.	Displays a date value as the full month name and the year in local format. If necessary, SAS will abbreviate the month name or use only numbers and delimiters for the month and year and/or a 2-digit year to fit the format width specified.	14	5–200
NLDATEYMM*w*.	Displays a date value as the local abbreviation for month name and the year. If necessary, SAS will use only numbers and delimiters for the month and year and/or a 2-digit year to fit the format width specified.	11	5–200
NLDATEYMS*w*.	Displays the month and year from a date value using only numbers and delimiters in local format. Will use 2-digit year if format width is 5 or 6.	7	5–200
NLDATEYQ*w*.	Displays a date value as calendar quarter and year. It is recommended that you use a minimum format width of 20 to ensure accurate output across all supported languages.	16	4–200

Format Name	Description	Default Width	Width Range
NLDATEYQL*w*.	Displays a date value as the full length for the calendar quarter and the year (for example, "3e trimestre 2014"). A width of 4 will display a 2-digit year. SAS will abbreviate as necessary to fit the format width specified.	18	4–200
NLDATEYQM*w*.	Displays a date value as a quarter abbreviation and the year (for example, "T3 2015"). SAS will use only numbers and delimiters to fit the format width if necessary. A width of 4 will display a 2-digit year.	7	4–200
NLDATEYQS*w*.	Will display the year and calendar quarter from a date value in local format using only numbers and delimiters. A width of 4 will display a 2-digit year.	6	4–200
NLDATEYR*w*.	Will display the 2- or 4-digit year from a date value.	16	2–200
NLDATEYW*w*.	Displays a date value as the week number and the year (for example, "Week 15 2014"). **Which week algorithm is used (U, V, or W) varies based on the value of the LOCALE= option.** See the WEEKU., WEEKV., and WEEKW. format discussions in Section 2.4.1, Date Formats, for more information about the week algorithms.	16	5–200

Table 6.7: NLS Datetime Formats

Format Name	Description	Default Width	Width Range
NLDATM*w.d*	Displays a datetime value as a datetime in local format.	30	10–200
NLDATMAP*w.d*	Displays a datetime value as month name, day, year, and time in local format. SAS will abbreviate as necessary to fit the format width specified and may substitute numbers and delimiters for the month name, day, and year.	32	16–200
NLDATMDT*w.d*	Displays the date from a datetime value with the month name, day, and year in local format. SAS will abbreviate the month name or substitute numbers and delimiters in local format for the month name, day, and year as necessary to fit the supplied format width.	18	10–100
NLDATML*w.d*	Displays the date from a datetime value with the month name, day, and year in local format. SAS will abbreviate the month name or substitute numbers and delimiters in local format for the month name, day, and year as necessary to fit the supplied format width.	30	9–200

Format Name	Description	Default Width	Width Range
NLDATMM*w.d*	Displays the date and time from a datetime value using the abbreviation for month name. SAS will use numbers and delimiters for the date if the format width is not wide enough and will further abbreviate the time to hours:minutes, then just hours if necessary.	24	9–200
NLDATMMD*w.d*	Displays the month and day from a datetime value. SAS will abbreviate the month name or use numbers and delimiters in local format (no year) if the format width cannot accommodate the full month name.	16	6–200
NLDATMMDL*w.d*	Displays the month and day from a datetime value. SAS will abbreviate the month name or use numbers and delimiters in local format (no year) if the format width cannot accommodate the full month name.	9	5–200
NLDATMMDM*w.d*	Displays the abbreviated month name and day from a datetime value. SAS will use numbers and delimiters in local format (no year) if the format width cannot accommodate the full month name.	9	5–200
NLDATMMDS*w.d*	Displays the month and day from a datetime value as numbers and delimiters only (for example, mm/dd or dd/mm).	5	5–200
NLDATMMN*w.d*	Displays the month name in local format from a datetime value. SAS will abbreviate if the full month name will not fit in the format width supplied.	9	4–200
NLDATMS*w.d*	Displays a datetime value as numbers and delimiters only, in local format (for example, 17/05/2014 13:47:06).	19	9–200
NLDATMTM*w.d*	Displays time of day from a datetime value in local time format.	16	16–200
NLDATMTZ*w.d*	Displays time of day from a datetime value in hours and minutes and the time zone offset for the locale.	32	16–200
NLDATMW*w.d*	Displays a datetime value as day of the week, date, and time in the local format. It is recommended that you use a minimum format width of 49 to ensure accurate output across all supported languages.	41	16–200
NLDATMWN*w.d*	Displays the day of the week from a datetime value in local format. SAS will abbreviate if the format width is too small to accommodate the full day of week name.	9	4–200

Format Name	Description	Default Width	Width Range
NLDATMWZ*w.d*	Displays day of the week and datetime in local format. SAS will abbreviate as necessary. It is recommended that you use a minimum format width of 55 to ensure accurate output across all supported languages.	40	16–200
NLDATMYM*w.d*	Displays the month name and year from a datetime value. SAS will abbreviate the month name and/or use a 2-digit year to fit the format width specified.	16	6–200
NLDATMYML*w.d*	Displays the month name and year from a datetime value. SAS will abbreviate the month name or use the month number, and use a 2-digit year if necessary to fit the format width specified.	14	5–200
NLDATMYMM*w.d*	Displays the abbreviated month name and year from a datetime value. SAS will use the month number and a 2-digit year if necessary to fit the format width specified.	11	5–200
NLDATMYMS*w.d*	Displays the month and year from a datetime value using numbers and delimiters only. SAS will use a 2-digit year if necessary to fit the format width specified.	7	5–200
NLDATMYQ*w.d*	Displays the year and quarter of the year from a datetime value in local format. SAS will abbreviate quarter and use a 2-digit year if necessary to fit the format width specified. It is recommended that you use a minimum format width of 20 to ensure accurate output across all supported languages.	16	4–200
NLDATMYQL*w.d*	Displays the year and quarter of the year from a datetime value in local format. SAS will abbreviate quarter, use numbers and delimiters only, and use a 2-digit year if necessary to fit the format width specified.	18	4–200
NLDATMYQM*w.d*	Displays the year and quarter of the year from a datetime value as an abbreviation in local format. SAS will use numbers and delimiters only and use a 2-digit year if necessary to fit the format width specified.	7	4–200
NLDATMYQS*w.d*	Displays the year and quarter of the year from a datetime value in local format as numbers and delimiters only. SAS will use a 2-digit year if necessary to fit the format width specified.	6	4–200
NLDATMYR*w.d*	Displays the year from a datetime value. SAS will use a 2-digit year if necessary to fit the format width specified.	16	2–200

Format Name	Description	Default Width	Width Range
NLDATMYW*w.d*	Displays a datetime value as the year and name of the week. SAS will abbreviate week and/or use a 2-digit year if necessary to fit the format width specified.	16	5–200
NLDATMZ*w.d*	Displays a datetime value as a datetime string in local format with the time zone offset.	40	16–200

Table 6.8: NLS Time Formats

Format Name	Description	Default Width	Width Range
NLTIMAP*w.d*	Displays a time value as the time, followed by AM or PM in local format. It is recommended that you use a minimum format width of 22 to ensure accurate output across all supported languages.	10	4–200
NLTIME*w.d*	Displays a time value as the time in local format.	20	10–200

Similar to the "EUR" informats, you can use NLS informats to process data according to the LOCALE= system option. Table 6.9 shows the NLS informats available, their default width specification, the width range, and the English language informat to which it is similar.

Table 6.9: NLS Time Informats

Category	Format Name	Description	Default Width	Width Range
Date	NLDATE*w.*	Reads local date strings of month name, day, and year in local format, or DDMMMYY(YY).	20	10–200
Datetime	NLDATM*w.*	Reads local datetime value strings.	30	10–200
Time	NLTIMAP*w.*	Reads local time strings containing AM and PM.	10	4–200
	NLTIME*w.*	Reads local time strings without AM and PM	20	10–200

The whole point of NLS formats and informats is that you do not have to worry about the specific language that will be using the format or informat. The LOCALE= system option will take care of it. In this way, the same SAS program can be used anywhere, and the output will be appropriate to the local language as long as the LOCALE= system option has been set correctly. It is important to understand that NLS formats and informats work just as well with the English language as with any other language, so there's no need to use SAS program code beyond the LOCALE= option to switch between English language formats/informats and other languages, unless you need a specific format that does not have an NLS equivalent. Also, bear in mind that any PICTURE formats you create for dates and times are NLS-compatible by default, and the display of date or time components such as month name are locale-sensitive.

6.5 NLS Date, Time, and Datetime Conversion Functions

There are three functions to help you work with date, time, and datetime values and provide output in the local language based on the LOCALE= system option in effect. The NLDATE(), NLTIME(), and NLDATM() functions take a date, time, or datetime value, respectively, and create a string variable according to a series of date and time directives. This is the same process that occurs if you create a custom format using the PICTURE statement (see Section 2.6), and then create a character variable by using the PUT function with your custom picture format and a date, time, or datetime value.

Example 6.14: Creating a Character Value Using a Custom Picture Format and the PUT() Function

```
PROC FORMAT;
PICTURE wordmonth (DEFAULT=15)
LOW-HIGH = '%B %Y' (DATATYPE=DATE);
RUN;

DATA picdate;
INPUT date :mmddyy10.;
month_and_year = PUT(date,wordmonth.);
FORMAT date date11.;
DATALINES;
05/15/2016
08/01/2012
10/31/2013
03/27/2014
;
RUN;

ODS RTF FILE="ex6.5.1.rtf";
PROC PRINT DATA=picdate NOOBS LABEL SPLIT='\';
label date="Original Date"
      month_and_year='Month and Year\Character Value'
;
RUN;
```

This is the resulting data set. Note that the numeric date variable, which has been formatted using the DATE. format, is right-justified, while the character string we created is left-justified.

Original Date	Month and Year Character Value
15-MAY-2016	May 2016
01-AUG-2012	August 2012
31-OCT-2013	October 2013
27-MAR-2014	March 2014

Essentially, the three NLS functions do the same thing, but without having to create the picture format using the FORMAT procedure. You put the description of the output string directly into the function as an argument, which is called the *descriptor*. The descriptor can also contain fixed text with the date directives, but this is somewhat counter to the purpose of these functions, as the fixed text you insert will not change with the LOCALE= option setting.

There are more date directives available with these functions than with the PICTURE statement of the FORMAT procedure. Following is the list of date directives you can use with the NLDATE(), NLTIME(), and NLDATM() functions. Although several of the directives may seem identical to those used with picture formats, these directives all insert a leading zero by default, so keep that in mind when using these functions. You will receive an error if you try to use a time directive with NLDATE(), or a date directive with NLTIME().

Table 6.10: NLDATE(), NLTIME(), and NLDATM() Date Directives

Date Directive	Description
#	removes the leading zero from the result.
%%	Specifies the percent (%) character.
%a	Locale's abbreviated (3-character) weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated (3-character) month name.
%B	Locale's full month name without padding. When you specify '%B %d,' there will always be one space between the month and day.
%C	Locale's full month name with blank padding. When you specify '%C %d,' there will be one or more spaces between the month and day, depending upon the length of the month's name.
%d	Zero-padded numerical day of the month. Use '#d' to suppress the leading zero.
%e	Blank-padded numerical day of the month. Use '#e' to suppress the leading blank.

Date Directive	Description
%F	Locale's full weekday name, padded with blanks. If you only want a single space between the weekday name and the next item, use '%A'.
%j	Zero-padded day of the year as a decimal number (001–366). Use '#j' to suppress any leading zeroes.
%H	Zero-padded hour (24-hour clock) as a decimal number (00–23). Use '#H' to suppress the leading zero.
%I	Zero-padded hour (12-hour clock) as a decimal number (01–12). Use '#I' to suppress the leading zero.
%m	Zero-padded month as a decimal number (01–12). Use '#m' to suppress the leading zero.
%M	Zero-padded minute as a decimal number (00–59). Use '#M' to suppress the leading zero.
%o	Blank-padded month as a decimal number (1–12). Use '#o' to suppress the leading blank.
%p	Locale's equivalent of a.m. or p.m.
%S	Zero-padded second as a decimal number (00–59). Use '#S' to suppress the leading zero.
%u	Weekday as a number in the range 1–7, where 1 is Monday and Sunday is 7.
%U	Zero-padded week-number-of-year (00–53) using the U algorithm, where Sunday is considered the first day of the week.
%V	Zero-padded week-number-of-year (00–53) using the ISO 8601-standard V algorithm, which defines the first week of the year as containing both January 4 and the first Thursday of the year. Therefore, if the first Monday of the year falls on January 2, 3, or 4, the preceding days of the calendar year are considered to be a part of week 53 of the previous calendar year.
%w	Weekday as a number in the range 0–6 where 0 is Sunday and 6 is Saturday.
%W	Zero-padded week-number-of-year (00–53) using the W algorithm, which uses Monday as the first day of the week.
%y	Zero-padded year without century as a decimal number (00–99).
%Y	Year with century as a decimal number (4-digit year). The year ranges from 1970 to 2069.

Here are the NLS date functions. When you are using text for the descriptor, the date directives *must* be enclosed in *single* quotes, or SAS will try to interpret them as macro calls.

Function Call	Explanation
NLDATE(*SAS-date-value,descriptor*)	Converts a SAS date value into a character string in the form described by *descriptor*, which is a combination of the directives in table 6.10 enclosed by single quotes. *descriptor* may also be a character variable containing a valid descriptor string.
NLDATM(*SAS-datetime-value,descriptor*)	Converts a SAS datetime value into a character string in the form described by *descriptor*, which is a combination of the directives in table 6.10 enclosed by single quotes. *descriptor* may also be a character variable containing a valid descriptor string.
NLTIME(*SAS-time-value,descriptor*)	Converts a SAS time value into a character string in the form described by *descriptor*, which is a combination of the directives in table 6.10 enclosed by single quotes. *descriptor* may also be a character variable containing a valid descriptor string.

Table 6.11 shows the results for different OPTIONS LOCALE= settings when you use the NLDATE function to create a character string containing the day-of-week name (%A), the three-letter abbreviated month name (%b), numerical day (%d), and four-digit year (%Y).

Table 6.11: The NLDATE Function

Sample Function Call	OPTIONS LOCALE=	Result
NLDATE('26OCT2014'd,'%A %b %d %Y');	de_DE	Sonntag Okt 26 2014
NLDATE('26OCT2014'd,'%A %b %d %Y');	en_US	Sunday Oct 26 2014
NLDATE('26OCT2014'd,'%A %b %d %Y');	es_MX	domingo oct 26 2014
NLDATE('26OCT2014'd,'%A %b %d %Y');	ru_RU	воскресенье окт. 26 2014
NLDATE('26OCT2014'd,'%A %b %d %Y');	zh_SG	星期日 10月 26 2014

Table 6.12 shows the results for different OPTIONS LOCALE= settings when you use the NLDATM function to create a character string containing the abbreviated day-of-week name (%a), numerical day without the leading zero (%#d), the four-digit year (%Y), and the colon-delimited 24-hour time with zero-padded hours and minutes (%H:%M).

Table 6.12: The NLDATM Function

Sample Function Call	OPTIONS LOCALE=	Result
NLDATM('06MAY2014:14:30:00'd,'%a %#d %B %Y %H:%M');	en_US	Tue 6 May 2014 14:30
NLDATM('06MAY2014:14:30:00'd,'%a %#d %B %Y %H:%M');	fr_FR	mar. 6 mai 2014 14:30
NLDATM('06MAY2014:14:30:00'd,'%a %#d %B %Y %H:%M');	ja_JP	火 6 5月 2014 14:30
NLDATM('06MAY2014:14:30:00'd,'%a %#d %B %Y %H:%M');	pl_PL	wt. 6 maja 2014 14:30
NLDATM('06MAY2014:14:30:00'd,'%a %#d %B %Y %H:%M');	sv_SE	tis 6 maj 2014 14:30

Table 6.13 shows the results for different OPTIONS LOCALE= settings when you use the NLTIME function to create a character string containing the colon-delimited 12-hour time with zero-padded hours and minutes (%I:%M) and the locale's AM/PM indicator (%p).

Table 6.13: The NLTIME Function

Sample Function Call	OPTIONS LOCALE =	Result
NLTIME('10:17:00'd,'%I:%M %p');	da_DK	10:17 f.m.
NLTIME('10:17:00'd,'%I:%M %p');	de_DE	10:17 vorm.
NLTIME('10:17:00'd,'%I:%M %p');	en_US	10:17 AM
NLTIME('10:17:00'd,'%I:%M %p');	es_SP	10:17 f.m.
NLTIME('10:17:00'd,'%I:%M %p');	zh_CN	10:17 上午

6.6 Date Formats and Informats for Other Calendars

SAS has the ability to handle dates from non-Julian calendars, such as Hebrew, Japanese, and Taiwanese. The date values continue to be stored as SAS dates, where January 1, 1960, is equal to zero, but these formats handle the conversion to the other calendars for the correct display of dates.

6.6.1 Hebrew Date Formats

HDATE*w*.

HDATE*w*. displays a SAS date value in Hebrew. You will need the correct character encoding installed on your system to display this correctly. The SAS date will be displayed as *yyyy mmmmm dd*, where *yyyy* is the year, *mmmmm* represents the month's name in Hebrew, and *dd* is the day-of-the-month. *w* can be from 9 to 17, with a default width of 17, and it is right-justified. Use odd numbers for *w* to get the best display.

HEBDATE*w*.

HEBDATE*w*. displays a SAS date value according to the Jewish calendar. It is a combined solar and lunar calendar. The Hebrew year is calculated by adding 3761 beginning in autumn of a specified year in the Gregorian calendar. *w* can be from 7 to 24, with a default width of 16, and it is right-justified. There are three forms of the display, long, default, and short, dependent upon the format width specified. Again, you will need the correct character encoding installed on your system, or you will get substitutions for nonprinting characters.

6.6.2 Japanese and Taiwanese Date Formats

MINGUO*w*.

MINGUO*w*. displays a SAS date value as a Taiwanese date value in the form *yy(yy)mmdd*, where *yy(yy)* is the year, *mm* is the number of the month, and *dd* is the day of the month. *w* can range from 1 to 10, with a default width of 8, and it is left-justified and zero-filled. The Taiwanese calendar uses 1912 as the base year (that is, 01/01/01 is January 1, 1912). Dates prior to this will display as a series of asterisks (*****). Also, the year values continue to increase past 100; they do not remain two-digit years and cycle, much like Julian dates. For example, January 1, 2012 is "100/01/01," not "00/01/01."

NENGO*w*.

NENGO*w*. writes a SAS date value in the form *e.yymmdd*, where *e* is the first letter of the name of the emperor (Meiji, Taisho, Showa, or Heisei), *yy* is the year, *mm* is the month, and *dd* is the day of the month. *w* can be from 2 to 10, with a default width of 10, and it is left-justified. SAS will omit the period if *w* isn't big enough.

6.6.3 Japanese and Taiwanese Date Informats

JDATEMYD*w*.

JDATEMYD*w*. allows you to convert Japanese Kanji in the form *yy(yy)mondd* to SAS date values, where *yy(yy)* is the year, *mon* is the Kanji representation of the name of the month, and *dd* represents the day of the month. *w* can be from 12 to 32, with a default width of 12. You can separate *(yy)yy*, *mon*, and *dd* with special characters or blanks, but you must make sure that the

width specification allows for any blanks and/or special characters in the input field. Two-digit years will be translated according to the YEARCUTOFF= option.

JNENGO*w*.

JNENGO*w*. reads Japanese Kanji date values in the form *yymmdd*, where *yy* is the year, *mm* is the Kanji representation of the name of the month, and *dd* represents the day of the month. Since *yy* is two digits long, this informat is always affected by the YEARCUTOFF= option. *w* can be from 16 to 32, with a default width of 16. You can separate *yy*, *mon*, and *dd* with special characters or blanks, but you must make sure that the width specification allows for any blanks and/or special characters in the input field.

MINGUO*w*.

MINGUO*w*. converts a Taiwanese date value into a SAS date value in the form *yy(yy)mmdd*, where *yy(yy)* is the year, *mm* is the number of the month, and *dd* is the day of the month. *w* can be from 6 to 10, with a default width of 6. You may use separators such as blanks, dashes, or slashes between the year, month, and day values, but they must be present between all of the values. The Taiwanese calendar uses 1912 as the base year (that is, 01/01/01 is January 1, 1912). In addition, the year values continue to increase past 100; they do not cycle. January 1, 2012, is "100/01/01," not "00/01/01." You will get a missing value if you use this format to read date strings where the year component is less than 1.

6.7 Other Software and Their Dates (Excel, Oracle, DB2)

Most software packages keep their dates in some sort of numerical form in much the same way that SAS does, while others have a special variable type for dates. Microsoft Excel stores dates as integers, but it uses January 1, 1900, instead of January 1, 1960, as day zero. Times are stored in Excel as fractions of days, so noon of a given day is .5 (exactly one-half of a day). Datetime values are stored in Excel as the day relative to 01/01/1900 plus the fraction of the day. In Excel, 6 p.m. on January 1, 1900, is represented as .75. Excel also has a major limitation on its date algorithm: It cannot store its dates as negative numbers. This means that any date prior to January 1, 1900, is going to be represented by a character string, not an Excel date value. Therefore, if you have historical dates in an Excel spreadsheet, you need to be aware that you will have to process any column(s) with historical dates as character columns, and use the INPUT() function to create your SAS date values. The ANYDT informats may also prove useful in situations like this. If you rely on an automated method of conversion from Excel to SAS, there is the possibility that a column with historical dates might be translated as a numeric column. If this occurs, then any dates (or datetimes) prior to January 1, 1900, will be missing in your SAS data set. You will run into the same problem exporting historical dates into Excel from SAS. You will have to export the column as a character column, and you won't be able to use any of the Excel date or math functions on cells containing them inside the spreadsheet.

These are conversion issues specific to Excel that may arise when you are trying to import or export data to or from Excel. When you import data from other software packages into SAS using

the IMPORT procedure, one of the database engines, or with pass-through SQL processing, SAS should understand and convert the dates, even though the reference date for the other software may differ. There are exceptions to this rule, one of which is using a pass-thru WHERE clause inside PROC SQL for foreign databases. You will have to know the date, time, or datetime format for the foreign database to select records based on dates, times, or datetimes. There are specific cases where SAS does not have an informat for certain datetime strings from other databases and cannot translate those values into their SAS equivalent. The general strategy to pursue in these cases is to parse the datetime string and work from there with a combination of informats and/or functions to create your SAS datetime values. If you will need to do this on a regular basis, you could create a macro or use PROC FCMP to process your troublesome datetime strings.

Sending dates to other databases and software packages should be fine if you use the EXPORT procedure or one of the database engines. If you are determined to send dates to another database or software package the hard way, then you will have to produce SAS date, time, or datetime values as character strings in the format of the other software and then import them using the methods available to other software packages. You can use a picture format and the PUT statement to accomplish this, as long as you know the correct representation of the package for which you are creating the data. For details on creating picture formats, see Section 2.6. Example 6.15 shows how this is done for a DB2 database.

Example 6.15: Writing Datetime Values for DB2 Using a Picture Format

```
PROC FORMAT;
PICTURE dbdate
LOW-HIGH = '%Y-%0m-%0d:%0H:%0M:%0S' (DATATYPE=DATETIME)
. - .Z = '0000-00-00:00:00:00';
RUN;

DATA _NULL_;
now = '01JUL2014:20:18:32'dt;
PUT "now displayed as datetime value: " @33 now;
PUT "now displayed as datetime19.: " @33 now DATETIME19.;
PUT "now displayed as dbdate.: " @33 now DBDATE.;
RUN;
```

The Result

```
now displayed as datetime value: 1719865112
now displayed as datetime19.:    01JUL2014:20:18:32
now displayed as dbdate.:        2014-07-01:20:18:32
```

6.7.1 The SASDATEFMT= System Option

This system option can be useful when you are working with one of the following databases: Aster, DB2 under UNIX and PC Hosts, Greenplum, Impala, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, PostgreSQL, SAP HANA, Sybase, Sybase IQ, Teradata, or

Vertica. It enables you to change the date format of a column in the DBMS. It is used to avoid data type mismatches between SAS date, time, or datetime values and the DBMS date columns when importing to SAS from a DBMS, or exporting to a DBMS from SAS. The syntax is as follows:

OPTIONS SASDATEFMT=(*DBMS-date-column$_1$ = SAS date-format$_1$, DBMS-date-column$_2$ = SAS date-format$_2$,… DBMS-date-column$_n$ = SAS date-format$_n$*);

DBMS-date-column is the name of a date column in the DBMS, and *SAS date-format* is the name of a SAS date, time, or datetime format. This format must also have an informat of the same name to work in this option. For example, DATE9. is both a format and an informat, so it would be valid.

In order for this option to have an effect, the DBMS column must have a type of DATE, TIME, or DATETIME; any other data type will be ignored by this option. You use this when the default SAS date format (which is DBMS- and data type-specific, see the SAS/ACCESS documentation for details) does not match the SAS date format you want or vice-versa.

6.8 Conclusion

In summary, when working with dates and times and datetimes from other software, you first have to make sure that you are not working with character strings masquerading as dates. If you have a character string, you will have to convert it to a SAS date, time, or datetime yourself with the INPUT() function (Section 3.3.3). The reverse also holds true when you are exporting to other databases. While most of the methods to traverse between different software and SAS will handle the import and export of dates, times, and datetimes accurately, it is always important to check the results at your destination.

Appendix A: A Quick Reference Guide to SAS Date, Time, and Datetime Formats

This table shows the result when the same date, time, or datetime value is displayed with the corresponding format, using the default length for the given format.

The reference date for this table is Thursday, September 18, 2014.

If You Want Your Date to Look Like This	Use This Format
18SEP14	DATE.
18	DAY.
18/09/14	DDMMYY.
18 09 14	DDMMYYB.
18:09:14	DDMMYYC.
18-09-14	DDMMYYD.
18092014	DDMMYYN.
18.09.14	DDMMYYP.
18/09/14	DDMMYYS.
Thursday	DOWNAME.
261	JULDAY.
14261	JULIAN.
09/18/14	MMDDYY.
09 18 14	MMDDYYB.
09:18:14	MMDDYYC.
09-18-14	MMDDYYD.
09182014	MMDDYYN.
09.18.14	MMDDYYP.
09/18/14	MMDDYYS.
09M2014	MMYY.
09:2014	MMYYC.
09-2014	MMYYD.

If You Want Your Date to Look Like This	Use This Format
092014	MMYYN.
09.2014	MMYYP.
09/2014	MMYYS.
September	MONNAME.
9	MONTH.
SEP14	MONYY.
3	QTR.
III	QTRR.
Thursday, September 18, 2014	WEEKDATE.
Thursday, 18 September 2014	WEEKDATX.
5	WEEKDAY.
2014-W37-05	WEEKU.
2014-W38-04	WEEKV.
2014-W37-04	WEEKW.
September 18, 2014	WORDDATE.
18 September 2014	WORDDATX.
2014	YEAR.
2014M09	YYMM.
2014:09	YYMMC.
2014-09	YYMMD.
14-09-18	YYMMDD.
14 09 18	YYMMDDB.
14:09:18	YYMMDDC.
14-09-18	YYMMDDD.
20140918	YYMMDDN.
14.09.18	YYMMDDP.
14/09/18	YYMMDDS.
201409	YYMMN.
2014.09	YYMMP.
2014/09	YYMMS.
2014SEP	YYMON.

If You Want Your Date to Look Like This	Use This Format
2014Q3	YYQ.
2014:3	YYQC.
2014-3	YYQD.
20143	YYQN.
2014.3	YYQP.
2014QIII	YYQR.
2014:III	YYQRC.
2014-III	YYQRD.
2014III	YYQRN.
2014.III	YYQRP.
2014/III	YYQRS.
2014/3	YYQS.
2014W37	YYWEEKU.
2014W38	YYWEEKV.
2014W37	YYWEEKW.

The reference time for this table is 2:45 p.m.

If You Want Your Time to Look Like This	Use This Format
14:35	HHMM.
15	HOUR.
875	MMSS.
14:35:00	TIME.
2:35:00 PM	TIMEAMPM.
14:35:00	TOD.

The reference datetime for this table is 7:20 p.m. on Friday, December 12, 2014.

If You Want Your Datetime to Look Like This	Use This Format
12DEC14:07:20:00 PM	DATEAMPM.
12DEC14:19:20:00	DATETIME.
12DEC14	DTDATE.
DEC14	DTMONYY.
Friday, 12 December 2014	DTWKDATX.
2014	DTYEAR.
14:4	DTYYQC.
12/12/2014 7:20 PM	MDYAMPM.

Appendix B: A Quick Reference Guide to NLS Date, Time, and Datetime Formats

This table shows the result when the same date, time, or datetime value is displayed with the corresponding format, using the default length for the given format.

The reference date for this table is Monday, July 7, 2014.

Format Name	Language		
	OPTIONS LOCALE= 'English_UnitedStates'	**OPTIONS LOCALE= 'Dutch_Belgium'**	**OPTIONS LOCALE= 'Russian_Russia'**
NLDATE.	July 07, 2014	07 juli 2014	07 июля 2014 г.
NLDATEL.	July 7, 2014	7 juli 2014	07.07.2014
NLDATEM.	Jul 7, 2014	7-jul.-2014	07.07.2014
NLDATEMD.	July 07	07 juli	07 июля
NLDATEMDL.	July 07	07 juli	07.07
NLDATEMDM.	Jul 07	07 jul.	07.07
NLDATEMDS.	07/07	07/07	07.07
NLDATEMN.	July	juli	Июль
NLDATES.	07/07/2014	07/07/2014	07.07.2014
NLDATEW.	Monday, July 7, 2014	maandag 7 juli 2014	пн, 7 июля 2014 г.
NLDATEWN.	Monday	maandag	Пн
NLDATEYM.	July 2014	juli 2014	*************** ❶
NLDATEYML.	July 2014	juli 2014	07.2014
NLDATEYMM.	Jul 2014	jul. 2014	07.2014
NLDATEYMS.	07/2014	07/2014	07.2014
NLDATEYQ.	3rd quarter 2014	Q3 2014	3-й кв. 2014

Format Name	Language		
	OPTIONS LOCALE= 'English_UnitedStates'	**OPTIONS LOCALE= 'Dutch_Belgium'**	**OPTIONS LOCALE= 'Russian_Russia'**
NLDATEYQL.	3rd quarter 2014	Q3 2014	3-й кв. 2014
NLDATEYQM.	Q3 2014	K3 2014	2014.3
NLDATEYQS.	2014/3	2014/3	2014.3
NLDATEYR.	2014	2014	2014
NLDATEYW.	Week 27 2014	Week 27 2014	Week 27 2014

➊-The Russian text is too long for the default format width.

The reference time for this table is 11:20.

Format Name	Language		
	OPTIONS LOCALE= 'English_UnitedStates'	**OPTIONS LOCALE= 'Dutch_Belgium'**	**OPTIONS LOCALE= 'Russian_Russia'**
NLTIMAP.	11:20 AM	11:20 AM	11:20 AM
NLTIME.	11:20:00	11:20:00 uur	11:20:00

The reference datetime for this table is Saturday, September 27, 2014, 05:45:00 p.m.

	Language		
Format Name	**OPTIONS LOCALE= 'English_UnitedStates'**	**OPTIONS LOCALE= 'Dutch_Belgium'**	**OPTIONS LOCALE= 'Russian_Russia'**
NLDATM.	27Sep2014:17:45:00	27 september 2014 17:45:00 uur	27.09.2014, 17:45:00
NLDATMAP.	September 27, 2014 05:45:00 PM	27 september 2014 05:45:00 PM	27.09.2014, 17:45:00
NLDATMDT.	September 27, 2014	27 september 2014	27.09.2014
NLDATML.	September 27, 2014 05:45:00 PM	27 september 2014 17:45:00	27.09.2014, 17:45:00
NLDATMM.	Sep 27, 2014 05:45:00 PM	27-sep.-2014 17:45:00	27.09.2014, 17:45:00
NLDATMMD.	September 27	27 september	27 сент.
NLDATMMDL.	September 27	27 september	27.09
NLDATMMDM.	Sep 27	27 sep.	27.09
NLDATMMDS.	09/27	27/09	27.09
NLDATMMN.	September	september	Сент.
NLDATMS.	09/27/2014 17:45:00	27/09/2014 17:45:00	27.09.2014 17:45:00
NLDATMTM.	17:45:00	17:45:00 uur	17:45:00
NLDATMTZ.	17:45:00 -0500	17:45:00 uur -0500	17:45:00 -0500
NLDATMW.	Saturday, September 27, 2014 05:45:00 PM	zaterdag 27 september 2014 17:45:00	сб, 27 сент. 2014 г., 17:45:00
NLDATMWN.	Saturday	zaterdag	Сб
NLDATMWZ.	Sat, Sep 27, 2014 05:45:00 PM -0500	za 27 sep. 2014 17:45:00 -0500	сб, 27 сент. 2014 г. -0500
NLDATMYM.	September 2014	september 2014	**************** ❶
NLDATMYML.	September 2014	september 2014	09.2014
NLDATMYMM.	Sep 2014	sep. 2014	09.2014
NLDATMYMS.	09/2014	09/2014	09.2014
NLDATMYQ.	3rd quarter 2014	Q3 2014	3-й кв. 2014
NLDATMYQL.	3rd quarter 2014	Q3 2014	3-й кв. 2014
NLDATMYQM.	Q3 2014	K3 2014	2014.3
NLDATMYQS.	2014/3	2014/3	2014.3
NLDATMYR.	2014	2014	2014
NLDATMYW.	Week 38 2014	Week 38 2014	Week 38 2014
NLDATMZ.	27Sep2014:17:45:00 -0500	27 september 2014 17:45:00 uur -0500	27.09.2014, 17:45:00 -0500

❶-The Russian text is too long for the default format width.

Appendix C: Troubleshooting Dates 101

This appendix is intended to be a quick solutions guide for *some* of the most common issues people have with dates and SAS. I hope to keep the discussion going online, because people often remark to me how difficult dates are in SAS, and they shouldn't be. You can find out more about this troubleshooting project on the author page for this book. Again, these are intended to be *simple* solutions to common beginner's questions.

Every person experiencing a problem with dates, times, or datetimes should immediately check to see if they are working with character values or numeric values. You can do this by using PROC CONTENTS, by looking at the properties in the Explorer window in interactive SAS, or by using the Data Set Attributes Task (under Tasks→Data) in SAS Enterprise Guide. Unless you are working with ISO 8601 durations and intervals, you should be working with numeric values. If not, then the first step you should take is to convert that character value into a proper SAS date, time, or datetime value.

Question 1: How do I convert my character value into a SAS date, time, or datetime value?

Case 1: If you are reading a flat file (CSV, tab-delimited, and so on), you will need to use the INPUT statement with the appropriate date, time, or datetime informat.

Sample Code

```
OPTIONS DATESTYLE=MDY;
DATA convert_char;
INFILE 'char_dates.txt' PAD MISSOVER;
INPUT id $ date1 :mmddyy10. date2 :date9. date3 :anydtdte.;
RUN;
```

Data File 'char_dates.txt'

```
101 04/17/2012 06dec2015 06272014
102 09/29/2014 15JUN2013 09feb2014
103 11/12/2013 3mar2015  08/19/2015
```

When you run the program with the data above, you will get the following data set. The date variables have intentionally been left unformatted so you can see that the data in the flat file has been turned into SAS date values. The ANYDTDTE. informat was used to read in the DATE3 variable to demonstrate how it can be used if you don't know what your character dates look like, or if they are in different date forms. Chapter 3 contains detailed descriptions of the informats that you can use in conjunction with the INPUT statement.

The Resulting Data Set

VIEWTABLE: Work.Convert_char

	id	date1	date2	date3
1	101	19100	20428	19901
2	102	19995	19524	19763
3	103	19674	20150	20319

Case 2: If you already have your data in a SAS data set, then you will have to use the INPUT function to translate your character dates into SAS dates.

Here is the data set we have with character dates:

VIEWTABLE: Work.Already_char

	id	date1	date2	date3
1	101	04/17/2012	06dec2015	06272014
2	102	09/29/2014	15JUN2013	09feb2014
3	103	11/12/2013	3mar2015	08/19/2015

Code to Convert Character Date Values into SAS Date Values

```
1.  OPTIONS DATESTYLE=MDY;
2.  DATA fix_char;
3.  set already_char;
4.  num_date1 = INPUT(date1,mmddyy10.);
5.  num_date2 = INPUT(date2,date9.);
6.  num_date3 = INPUT(date3,anydtdte11.);
7.  RUN;
```

The INPUT function is used in lines 4, 5, and 6 to create the new variables num_date1, num_date2, and num_date3. The reason we are creating new variables is that you cannot change an existing variable from character to numeric, so you cannot use the variable names date1, date2, or date3 because they exist in the data set ALREADY_CHAR. The INPUT function uses the informats detailed in Chapter 3, just like the INPUT statement.

The Resulting Data Set

VIEWTABLE: Work.Fix_char

	ID	date1	num_date1	date2	num_date2	date3	num_date3
1	101	04/17/2012	19100	06dec2015	20428	06272014	19901
2	102	09/29/2014	19995	15JUN2013	19524	09feb2014	19763
3	103	11/12/2013	19674	3mar2015	20150	08/19/2015	20319

In general, when you are dealing with SAS dates, times, and datetimes, you should be working with numeric variables. The exception to this rule is ISO durations and intervals, which is covered in Chapter 4.

Question 2: Why do I get the log message "Variable xxxxxx has been defined as both character and numeric"?

You are trying to convert a character variable into a SAS date, but you're using the same variable name in the INPUT function and for the result.

```
date = INPUT(date,mmddyy10.);
```

Instead, you must use a different variable name to store the result of the INPUT function.

```
new_date = INPUT(date,mmddyy10.);
```

Question 3: Why do I get the log message "NOTE: Invalid argument to function INPUT" when I try to use the INPUT function to convert a character date into a SAS date?

You are using the wrong informat for the characters you are converting. As an example, if your character values look like "04MAY2015," you'll get this error if you don’t use the DATE. (or ANYDTDTE.) informat.

Quickest fix: Use one of the "ANYDATE" informats. Read Section 3.4.4 for details and the possible pitfalls of this strategy.

Best fix: Find the correct informat that fits the character string you're converting.

Question 4: Why do I get errors when I am reading a flat file with dates, and I know I'm using the correct informat?

This is most often an issue of not reading enough characters in the field, or of reading too many.

Best fix: Make sure you specify the correct length on your informat, and use the colon (:) modifier for your informat (see Example 3.2).

Question 5: Why doesn't my date comparison work?

This is generally a problem when you are importing data from another database. First, ensure that the variables you are working with are numeric. Second, many databases store their dates as datetime values, so even if using dates works in the database, it will not work when you try to compare that with a SAS date value.

Best fix: Once you've verified that your data aren't character, try using the DATEPART() function (Table 5.3) on the variable you've imported and compare that with your SAS date.

Question 6: My date values look right, so why can't get I my date comparison to work?

This frequently means that one of your values is character. Removing formats from a SAS date value is a good way to make sure that you've got a SAS date value. As an example, if the date is in 2014, it should be a number between 19724 and 20088.

Question 7: How can I subset my data before or after a specific date/time/datetime?

You have to use a date literal such as '01JAN2014'd, a time literal such as '14:00't, or a datetime literal such as '17OCT2014:07:00'dt.

```
DATA junestock;
SET sashelp.citiday;
WHERE date BETWEEN '01JUN1988'd AND '30JUN1988'd;
RUN;
```

This is one way to create a subset of data for June 1998 from the data set SASHELP.CITIDAY. However, don't let the format of the date fool you. Here's the same method applied to the data set SASHELP.CITIMON.

```
DATA stock8081;
SET sashelp.citimon;
WHERE date BETWEEN '01JAN1980'd AND '31DEC1981'd;
RUN;
```

Even though the dates are monthly, they are still SAS dates, so you have to specify the boundaries as complete SAS dates, not just a month and year.

Question 8: How do I get a date from a date and time?

Use the DATEPART() function after you've converted it to a SAS datetime value. If you just need the time, use the TIMEPART() function. This example takes a datetime value and separates it into date and time, formatted and unformatted versions.

```
DATA date_from_datetime;
datetime = '8APR2014:16:00'dt;
fmt_datetime = datetime;
date = DATEPART(datetime);
fmt_date = date;
time = TIMEPART(datetime);
fmt_time = time;
FORMAT fmt_datetime mdyampm21. fmt_date mmddyy10. fmt_time
timeampm.;
RUN;

PROC PRINT DATA=date_from_datetime NOOBS;
RUN;
```

datetime	fmt_datetime	date	fmt_date	time	fmt_time
1712592000	4/8/2014 4:00 PM	19821	04/08/2014	57600	4:00:00 PM

Question 9: Why do my dates look like a bunch of numbers that don't make any sense, and how can I fix it?

If your date value looks like a number such as 16789, then it is displaying as its SAS date, and all you need to do is to format it using one of the date formats in Appendix A, or if you need one of the NLS formats to translate the date into a language other than English, see Section 6.4 and Appendix B.

Question 9a: I formatted my date and now all I get is a bunch of asterisks (************). What's wrong?

If your unformatted SAS value is more than 7 digits long, or you've used a date format and you get asterisks, you probably have a datetime value, and you need to use a datetime format. See Appendix A for the quick list.

Question 10: My date looks like "06/25/2014," but I need to make it look like "2014/06/25." I know it's a SAS date value in the data set, so how do I change it?

Use the FORMAT statement. It does not matter what format the variable has associated with it in the data set. As long as it is a SAS date, time, or datetime value, you can change the display by changing the format in the procedure where you are displaying the variable. However, if you get an error message along the lines of "Format $... not found," the dollar sign ($) is telling you that the variable you are trying to format is a character variable. In that case you will have to convert it to the corresponding SAS value.

Let's print out some records from a data set using this code:

```
ODS RTF FILE="apxc_10.rtf";
PROC PRINT DATA=book.dailysales (OBS=5);
VAR date;
RUN;
```

Obs	date
1	22MAY2014
2	23MAY2014
3	24MAY2014

Obs	date
4	25MAY2014
5	26MAY2014

What you see in the date column is just a SAS date that has been associated with the DATE9. format when the data set was created, so without any FORMAT statement in the PRINT procedure, that is what it looks like. Now we'll add a FORMAT statement.

```
1   PROC PRINT DATA=book.dailysales (OBS=5);
2   VAR date;
3   FORMAT date yymmdds10.;
4   RUN;
```

Obs	date
1	2014/05/22
2	2014/05/23
3	2014/05/24
4	2014/05/25
5	2014/05/26

And that's how you change the way a date, time, or datetime looks in output from its default format.

Question 11: My date is numeric, like 20140815. How do I get it to display as 08/15/2014?

This is usually the result of an import that went wrong. First, make sure that it really is the number 20,140,815. If this is true, then you can turn that number into a character string with the PUT function and convert it to a date value like you do with any other character date string. Here's one way:

```
DATA c11;
bad_date = 20140815;
char_date = PUT(bad_date,10.);
SAS_date = INPUT(STRIP(char_date),yymmdd8.);
fmt_SAS_date = SAS_date;
FORMAT fmt_SAS_date weekdate.;
RUN;
```

```
PROC PRINT DATA=c11 NOOBS;
RUN;
```

bad_date	char_date	SAS_date	fmt_SAS_date
20140815	20140815	19950	Friday, August 15, 2014

Question 12: How can I display just the month and year, when I have month, day, and year?

By definition, SAS dates are a specific day in a specific month during a specific year, so you can't just delete the day and store the month and year. If you want to display the value as only a month and a year, then just use a format that only displays the month and year.

Question 13: How do I convert from datetime format to date format?

In order to answer this question, you need to know what you want. If you need a SAS date value, then you need to convert it from seconds since midnight, January 1, 1960, to days since January 1, 1960, by using the DATEPART function. If you just want it to display like a date, and don't need or want the extra variable in your data set, then you can use a format. See Appendix A for the available SAS datetime formats and the output they produce.

```
DATA q12;
datetime = '31AUG2014:10:15'dt;
fmt_datetime = datetime;
date_from_datetime = DATEPART(datetime);
fmt_date = date_from_datetime;
FORMAT fmt_datetime dtwkdatx. fmt_date weekdatx.;
RUN;

PROC PRINT DATA=q12 NOOBS;
RUN;
```

Note the difference in the actual SAS values. Even though the formatted values are identical, the variable *datetime* is a SAS datetime value, while the variable *date_from_datetime* is a SAS date value created with the DATEPART function.

datetime	fmt_datetime	date_value_from_datetime	fmt_date
1725099300	Sunday, 31 August 2014	19966	Sunday, 31 August 2014

Question 14: I can't find a SAS format to make my date/time/datetime look the way I want. OR: I need to output my date so it looks like . . .

First, check Appendix A and the SAS documentation to see if you can find a format that matches your needs. If not, then use the FORMAT procedure with the PICTURE statement. Section 2.7 of the book provides all the details you need to create a custom format, including the SAS date directives that tell SAS what you want your date, time, or datetime to look like. Don't worry that you may be duplicating an existing SAS format; that will not cause a problem as long as you do not use the same name as a SAS-supplied format. You will see a note in the log, and your custom format will not be created.

Question 15: How can I read a date/time/datetime formatted like . . .?

The simple answer is to use one of the "ANYDATE" informats. Don't forget about the DATESTYLE= system option, and check your results carefully before using them. The "ANYDATE" informats are easy, but they can be fooled, even when four-digit years are used. See Section 3.4.4 for details.

Question 16: How can I convert a UTC time value to a specific time zone?

If your "time value" is a datetime, use the TZONEU2S() function. This changes the SAS value, so don't overwrite your original value unless this is what you want to do. If you have just a time value, you will need to get the offset via the TZONEOFF() function, which will return the offset from GMT for the time zone you supply.

```
DATA q16;
datetime = "26MAR2014:11:47:00"dt;
gmt_datetime = TZONES2U(datetime);
RUN;

PROC PRINT DATA=q16 NOOBS;
FORMAT datetime gmt_datetime mdyampm21.;
RUN;
```

datetime	gmt_datetime
3/26/2014 11:47 AM	3/26/2014 4:47 PM

Question 17: How can I make a single date variable from separate month, day, and year?

The MDY function will let you supply a numeric month, day, and year, and it will calculate the SAS date value. You see this often when reading Excel files, where the month, day, and year are in separate columns. We'll use in-stream comma-separated data instead of an Excel file for the example.

```
1  DATA Q17;
2  INFILE DATALINES DLM=',';
3  INPUT month day year;
4  SAS_date = MDY(month,day,year);
5  fmt_date = SAS_date;
6  FORMAT fmt_date mmddyyd10.;
7  DATALINES;
8  2,15,2013
9  9,6,2014
10 12,17,2013
11 5,22,2014
12 ;;;;
13 RUN;

14 PROC PRINT DATA=q17 NOOBS;
15 RUN;
```

The **SAS_date** column shows the SAS date value resulting from the use of the MDY function in line 4.

month	day	year	SAS_date	fmt_date
2	15	2013	19404	02-15-2013
9	6	2014	19972	09-06-2014
12	17	2013	19709	12-17-2013
5	22	2014	19865	05-22-2014

Question 18: I have a date and a time. How can I make a SAS datetime from them?

This occurs when you import data into SAS and the date is in one column and the time is in another. The DHMS() function is what is used to create datetimes from dates and times. Although you might think that you have to break the time value into hours, minutes, and seconds to use the function, remember that SAS times are maintained in seconds since midnight, so you can leave hours and minutes set to zero and put the time variable into the seconds parameter. We will use in-stream data for the example. Pay attention to line 3. This is how you create a datetime from a SAS date and a SAS time.

```
1  DATA q18;
2  INPUT date : yymmdd10. time : time.;
3  datetime = DHMS(date,0,0,time);
4  fmt_date = date;
5  fmt_time = time;
6  fmt_datetime = datetime;
7  FORMAT fmt_date mmddyy10. fmt_time timeampm. fmt_datetime
   mdyampm21.;
8  DATALINES;
9  2014-8-17 5:45
10 2014-03-11 9:06
11 2015-02-27 15:43
12 2014-12-13 10:00
13 ;
14 RUN;

15 ODS RTF FILE='apxc_18.rtf';
16 PROC PRINT DATA=q18 NOOBS;
17 RUN;
```

date and *time* are the SAS values as read from the data, and *datetime* is the SAS value created in line 3 of the above code. We have created duplicate variables for formatting.

date	time	datetime	fmt_date	fmt_time	fmt_datetime
19952	20700	1723873500	08/17/2014	5:45:00 AM	8/17/2014 5:45 AM
19793	32760	1710147960	03/11/2014	9:06:00 AM	3/11/2014 9:06 AM
20146	56580	1740670980	02/27/2015	3:43:00 PM	2/27/2015 3:43 PM
20070	36000	1734084000	12/13/2014	10:00:00 AM	12/13/2014 10:00 AM

Question 19: How can I calculate the number of days between two dates?

There are multiple ways to calculate the difference between two dates, times, or datetimes in SAS. This is one of the best things about having them represented as numbers. The simplest way to find the difference is to subtract one date from another. It is important that you make sure that you are subtracting dates from dates, times from times, and datetimes from datetimes.

Question 20: How do I convert a numeric SAS date value into character format?

The first question you should ask yourself is why. Formatting the SAS date value will give you the display you want. Even when you are exporting to other software, as long as you put the date into a form that the other software understands, it should work fine. You don't need to do the character conversion, and you will not be able to do any calculations with that character variable. The only time that I would convert a date value into a character value is when I need to put the date inside of

a long string of text such as, "The subject started the trial on January 15, 2012, and mastered task A within 15 minutes." It doesn't matter if a format is permanently associated with the date variable or what it looks like when you look at the data set; remember that you can change the way a date is displayed in any procedure where you can use a FORMAT statement.

Here's a data set with a variable that has been formatted with the WORDDATX. format. This means that anytime the variable DATE is displayed in a SAS procedure, it will be displayed with the WORDDATX. format as shown below.

VIEWTABLE: Work.Q20

	date
1	30 June 2014
2	18 September 2014
3	14 March 2014
4	18 July 2014
5	26 April 2014

Now, let's run a PROC PRINT, but I want to show the date as DD-MMM-YYYY, so I use the FORMAT statement to change the display for this run:

```
PROC PRINT DATA=q20 NOOBS;
FORMAT date date10.;
RUN;
```

Formatted with DATE10.

date
30JUN2014
18SEP2014
14MAR2014
18JUL2014
26APR2014

What if you want to see it as YYYY-MM-DD?

```
PROC PRINT DATA=q20 NOOBS;
FORMAT date yymmddd10.;
RUN;
```

Formatted with YYMMDDD10. (forces the separator to be the dash)

date
2014-06-30
2014-09-18
2014-03-14
2014-07-18
2014-04-26

But when we remove all the formats by using a FORMAT statement without a format name, we see the actual SAS date value:

```
PROC PRINT DATA=q20 NOOBS;
FORMAT date;
RUN;
```

Format Removed

date
19904
19984
19796
19922
19839

If you still want to convert your SAS date into a character value, then you can use the PUT function. Line 2 is a precaution to make sure that the character variable is long enough to hold the longest value. Note that you have to use a different name for the character variable you create in line 4.

```
1.  DATA q20_plus;
2.  LENGTH chardate $ 31;
3.  SET q20;
4.  chardate = PUT(date,weekdate.);
5.  RUN;
```

chardate	date
Monday, June 30, 2014	19904
Thursday, September 18, 2014	19984
Friday, March 14, 2014	19796
Friday, July 18, 2014	19922
Saturday, April 26, 2014	19839

Index

Symbols and Numerics

& (ampersand) 192
= (equals sign) 207
$ (dollar sign) 61
%% date directive 49

A

%a date directive 49
%A date directive 49
AFR language prefix 214
ampersand (&) 192
ANYDTDTE*w.* informat 82–83
  ANYDTDTM*w.* informat and 84–85
  ANYDTTME*w.* informat and 85–86
  DATESTYLE= system option and 81, 82–83, 85–86
  troubleshooting 239
ANYDTDTM*w.* informat 81, 84–85
automatic macro variables 186–189

B

%b date directive 49
%B date directive 49
B8601CI*w.d* informat 78, 108
B8601DA*w.* format 93
B8601DA*w.* informat 104
B8601DJ*w.d* informat 79, 108
B8601DN*w.* format 99
B8601DT*w.d* format 99–100
B8601DT*w.d* informat 109
B8601DX*w.d* format 100–101
B8601DZ*w.d* format 102
B8601DZ*w.d* informat 110
B8601LZ*w.d* format 96
B8601TM*w.d* format 94
B8601TM*w.d* informat 105
B8601TX*w.d* format 94–95
B8601TZ*w.d* format 97–98
B8601TZ*w.d* informat 105–106

C

calculations
  INTCK() function and 151–156
  INTNX() function and 156–159
  number of days between dates 145–149
  number of years between dates 146–147
CALL IS8601_CONVERT 123–136
CALL SYMGET() function 192–193
CALL SYMPUT() function 192–193
case sensitivity 48–49
CAT language prefix 214
CATS() function 14
CATT() function 14
CATX() function 14
character constants 61
character strings
  DATETIME informat and 63
  PUT() function and 55–56
character variables
  INPUT() function and 61–62
  INPUTC() function 61–62
  PUT() function and 55–56
  PUTN() function and 55–56
COMPRESS() function 14
constants, date and time as 3–5
CONVERT statement, EXPAND procedure
  converting to higher frequency 202–203
  METHOD= option 206
  OBSERVED= option 206–212
CRO language prefix 214
CSY language prefix 214

D

%d date directive 49, 51
DAN language prefix 214
DATA step
  FORMAT statement 10–137
  %LET statement and 191, 192
DATADIF() function 145–146
DATALINES statement 61
DATATYPE= option 48, 51

date directives, picture format 50
DATE() function 187–188
&DATE macro variable 187–188
DATE system option 7
DATEAMPM*w.d* format 42–43, 233
DATEPART() function 140
- datetime formats and 41–42
- troubleshooting 241, 242–243, 245

dates
- automatic macro variables 186–189
- CALL SYMPUT() function and 192–193
- as constants 3–5
- counters for 1
- creating character strings 55–56
- custom formats 47–55
- datetime values and 13–14, 41–42
- default justification 13
- Excel and 227–228
- external representation of 2–3
- formats for 14–37, 92–103
- graphing 194–200
- Hebrew formats 226
- informats for 59, 61, 64–73, 81–86, 103–111
- internal representation of 2
- international formats and informats 212–220
- interval definitions 149–151
- interval functions 151–159
- Japanese formats 226
- Japanese informats 226–227
- quick reference 231–233, 235–237
- shifting intervals 159–168
- %SYSFUNC() macro function and 187–189
- Taiwanese formats 226
- Taiwanese informats 226–227
- in titles 186–187
- troubleshooting 239–251
- width specification 5–6, 13
- YEARCUTOFF= system option and 57–59

DATESTYLE= system option
- ANYDTDTE*w.* informat and 82–83, 85–86
- ANYDTDTM*w.* informat and 84
- LOCALE= system option and 212
- missing values example 81
- troubleshooting 246

DATESTYLE=DMY system option 81–82
DATESTYLE=LOCALE system option 81
DATESTYLE=MDY system option 81–82
DATESTYLE=YMD system option 81–82
DATETIME() function 138
datetime values
- automatic macro variables and 186–189
- CALL SYMPUT() function and 192–193
- as constants 3–5
- custom formats 47–55
- date formats and 13–14, 41–42
- default justification 13
- Excel and 227–228
- external representation of 2–3
- formats for 41–46, 99–103
- informats for 61, 78–81, 81–82, 108–111
- internal representation of 2
- international formats and informats 212–220
- interval definitions 150–151
- interval functions 151–159
- quick reference 234, 235–237
- shifting intervals 159–1682
- %SYSFUNC() macro function and 187–189
- width specification 5–6, 37, 41–42
- YEARCUTOFF= system option and 57–59

DATETIME*w.* informat
- ANYDTDTE*w.* informat and 82
- ANYDTDTM*w.* informat and 84
- ANYDTTME*w.* informat and 85–86
- character strings and 63

DATETIME*w.d* format 43–44, 79, 213, 233
DATE*w.* format 15
- DTDATE*w.* format and 44
- international format for 213
- quick reference 231

DATE*w.* informat 64
- ANYDTDTE*w.* informat 82
- ANYDTDTM*w.* informat 84
- ANYDTTME*w.* informat 85–86

DATJUL() function 140–141
DAY() function 138
DAY interval 150
- INTCK() function 152
- INTNX() function 158
- shift point 160

DAY*w.* format 15, 231

DB2 databases 227–228
DDMMYYB. format 17, 231
DDMMYYC. format 17, 231
DDMMYYD. format 17, 231
DDMMYYN. format 231
DDMMYYP. format 17, 231
DDMMYYS. format 231
DDMMYY*w*. format 15–16, 30–31
    international format for 213
    quick reference 231
    YYMMDD*w*. format and 30–31
DDMMYY*w*. informat 64–65
    ANYDTDTE*w*. informat and 82
    ANYDTDTM*w*. informat and 84
    ANYDTTME*w*. informat and 85–86
    DATASTYLE= system option 81–82
DDMMYY*xw*. format 16–17
    MMYY*xw*. format and 20
    YYMM*xw*. format and 30
    YYQ*xw*. format and 33–34
DES language prefix 214
DEU language prefix 214
DFLANG= system option 212
DHMS() function 141–142, 247–248
dollar sign ($) 61
dot (.)
    See period
DOWNAME*w*. format 17
    international format for 213
    quick reference 231
DTDATE9. format 44
DTDATE*w*. format 44
    DATE*w* format and 15
    quick reference 233
DTDAY interval 150
    INTCK() function 155
    INTNX() function 158
    shift point 160
DTHOUR interval 151, 161
DTMINUTE interval 151, 161
DTMONTH interval 151
    INTNX() function 158
    shift point 161
DTMONYY*w*. format 44
    MONYY*w*. format and 21
    quick reference 233
DTQTR interval 151
    INTNX() function 157, 158
    shift point 161
DTRESET system option 7
DTSECOND interval 151, 161
DTSEMIMONTH interval 151
    INTNX() function 158
    shift point 161
DTSEMIYEAR interval 151
    INTNX() function 157, 158
    shift point 161
DTTENDAY interval 151
    INTNX() function 158
    shift point 161
DTWEEK interval 150
    INTNX() function 157
    shift point 161
DTWEEKDAY interval 151
    INTNX() function 157
    shift point 161
DTWKDATX*w*.format 45
    quick reference 233
DTYEAR interval 151
    INTNX() function 157, 158
    shift point 161
DTYEAR*w*. format 45, 233
DTYYQC*w*. format 46, 233
durations, ISO 8601 116–136

E

E8601DA*w*. format 93
E8601DA*w*. informat 104
E8601DN*w*. format 99
E8601DT*w.d* format 100
E8601DT*w.d* informat 109–110
E8601DX*w.d* format 101–102
E8601DZ*w.d* format 102–103
E8601DZ*w.d* informat 110–111
E8601LZ*w.d* format 96–97
E8601LZ*w.d* informat 107
E8601TM*w.d* format 94
E8601TM*w.d* informat 105
E8601TX*w.d* format 95–96
E8601TZ*w.d* format 98–99

E8601TZ*w.d* informat 106–107
equals sign (=) 207
ERROR automatic variable 6–63
ESP language prefix 214
"EUR" formats 213–214
"EUR" informats 213–214
EURDFDD. format 213
EURDFDE*w*. format 213
EURDFDN. format 213
EURDFDT*w*. format 213
EURDFDWN. format 213
EURDFMN. format 213
EURDFMY*w*. format 213
EURDFWDX. format 213
EURDFWKX. format 213
Excel (Microsoft) 227–228
EXPAND procedure
    capabilities 200–202
    CONVERT statement 206–212
    converting to higher frequency 202–203
    FROM= option 206, 209–212
    ID statement 204–205
    interpolating missing values 205–206
    TO= option 206, 209–212
external representation, date and time 2–3

F

%F date directive 223
FCMP procedure 52–55
FIN language prefix 214
FOOTNOTE statement 186
FORMAT procedure
    PICTURE statement 47–52
    troubleshooting 246
    VALUE statement 47–48
FORMAT statement
    date directives and 50
    functionality 10–13
    INFORMAT statement and 59–60
    troubleshooting 243–244
formats 10–13, 59
    custom 47–55
    date constants and 4–5
    for dates 14–37
    for datetime 41–46
    datetime values 99–103
    "EUR" 213–214
    external representation of date, time and 2–3
    graphing dates and 194–200
    Hebrew 226
    ISO 8601 92–103, 117–121
    Japanese 226
    "NLS" 214–220
    PUT() function and 55–56
    quick reference 231–233
    Taiwanese 226
    for time 37–41
    using wrong 62–63
FRA language prefix 214
FROM= option, EXPAND statement 206, 209–212
FRS language prefix 214
FUNCTION statement 52–55
functions
    calculating intervals 145–149
    creating date, time 140–145
    current date, time 137–138
    extraction 138–140

G

graphing dates 194–200
Gregorian year
    JULIAN*w*. informat 65
    PDJULG4. informat 66
    PDJULG*w*. informat 22
    PDJULI*w*. format 22
    PDJULI*w*. informat 66

H

%H date directive 49, 223
HDATE*w*. format 226
HEBDATE*w*. format 226
Hebrew date formats 226
HHMMSS*w*. informat 73–75
HHMM*w.d* format 38, 233
HMS() function 143
HOLIDAY() function 148–149, 170
HOUR() function 140
HOUR interval 151
    INTNX() function 158
    shift point 161

HOUR*w.d* format 38–39, 233
HUN language prefix 214

I

%I date directive 49, 223
ID statement, EXPAND procedure 204–205
imputed dates 52–55
INFORMAT statement 59–61, 59–63
informats
  about 59
  ANYDT variants 81–86
  "EUR" 213–214
  for dates 64–73
  for datetime 78–81
  for datetime values 108–111
  for time 73–78
  Hebrew 226
  INFORMAT statement 59–63
  ISO 8601 103–111
  ISO 8601 duration and interval 121–123
  Japanese and Taiwanese 226–227
  "NLS" 220
  using wrong 62–63
INPUT() function
  functionality 3
  informats and 59, 61–62
  troubleshooting 240, 241
INPUT statement 60–61
INPUTC() function 61–62
INPUTN() function 61–62
INTCINDEX() function 183
INTCK() function 159, 161–162
  calculating intervals 167–168
  WORKINGDAYS interval 175–176
INTCYCLE() function 183
internal representation, date and time 2
interval multipliers
  graphs and 194–200
  shifting intervals and 162–163
intervals
  basics of 149–151
  creating 169–176
  custom 162–163
  INTCK() function 151–156
  interval functions 176–181
  INTNX() function 156–159
  ISO 8601 116–136
  measuring 167–168
  number of days between dates 145–149
  number of years between dates 146–147
  retail calendar 181–183
  shifting 159–168
INTFIT() function 177–178
INTFMT() function 178–179
INTGET() function 179–180
INTINDEX() function 183
INTNX() function 151, 156–160, 167–168
INTSEAS() function 183
INTSHIFT() function 180–181
INTTEST() function 181
ISO 8601 91–92
  durations and intervals 116–136
  formats 92–103
  informats 103–111
ITA language prefix 214

J

%j date directive 49, 223
Japanese date formats 226
Japanese date informats 226–227
JDATEMYD*w.* informat 226–227
Jewish calendar 226
JNENGOW. informat 227
Joshi, Bhairav 2
JULDATE() function 138
JULDATE7() function 138
JULDAY*w.* format 17–18, 233
Julian date
  JULDATE() function 138
  JULDATE7() function 138
  JULDAY*w.* format 17–18
  JULIAN*w.* format 18
  JULIAN*w.* informat 65
  PDGJULI1. format 23
  PDJULG. informat 66
  PDJULG4. informat 66
  PDJULG*w.* format 22
  PDJULI*w.* format 22–23
  PDJULI*w.* informat 66
JULIAN*w.* format 231

JULIAN*w.* informat 65
  ANYDTDTE*w.* informat and 82
  ANYDTDTM*w.* informat and 84
  ANYDTTME*w.* informat and 85–86
justification
  date formats 13–14
  PDJULI*w.* format and 22

K

Kanji representation 227

L

%LEFT() macro function 192
LENGTH statement 5–6
%LET statement 191, 192
LISTING destination 13–14
literal values
  quotation marks and 3
  YEARCUTOFF= system option and 57–59
LOCALE= system option 111, 214–215

M

%m date directive 49, 223
%M date directive 49
MAC language prefix 214
macro functions, date and time in 186–189
macro variables
  CALL SYMPUT() function and 192–193
  dates and 185–193
  quotation marks and 186
MAKEDATE() function 52–55
MDY() function 143, 247
MDYAMPM*w.* format 46, 233
MDYAMPM*w.d* informat 79–80
METHOD= option, CONVERT statement (EXPAND) 206
Microsoft Excel 227–228
MINGUO*w.* format 226
MINGUO*w.* informat 227
MINUTE() function 140
MINUTE interval 151
  INTNX() function 158
  shift point 161
missing values
  DATESTYLE= system option 81
  EXPAND procedure and 205–206
  symbol for 50
  wrong informats and 62–63
MMDDY*w.* format 18
  MMDDYY*xw.* format and 19
  quick reference 231
  YYMMDD*w.* format and 30–31
MMDDYY10. format 11–13, 18
MMDDYYB. format 19, 231
MMDDYYC. format 19, 231
MMDDYYD. format 231
MMDDYYN. format 231
MMDDYYP. format 19, 231
MMDDYYS. format 231
MMDDYY*w.* informat 65–66
  ANYDTDTE*w.* informat and 82
  ANYDTDTM*w.* informat and 84
  DATESTYLE= system option and 81–82
MMDDYY*xw.* format 19
  MMYY*xw.* format and 20
  YYMM*xw.* format and 30
  YYQ*xw.* format and 33–34
MMSS*w.d* format 39, 233
MMYYC. format 20, 231
MMYYD. format 20, 231
MMYYN. format 20, 232
MMYYP. format 20, 232
MMYYS. format 232
MMYY*w.* format 19–20, 231
MMYY*xw.* format 20
MONNAME*w.* format 14, 21
  international format for 213
  quick reference 232
MONTH() function 138
MONTH interval 150
  example 202–203
  INTCK() function 152
  INTNX() function 158
  shift point 160, 165–166
MONTH*w.* format 21, 232
MONYY*w.* format 21
  DTMONYY*w.* format and 44
  international format for 213
  quick reference 232

MONYY*w*. informat 66
  ANYDTDTE*w*. informat and 82
  ANYDTDTM*w*. informat and 84
  ANYDTTME*w*. informat and 85–86
  explanation of 66
MSEC8. informat 76

N

$N8601BA*w.d* format 119
$N8601Bw. informat 121–122
$N8601B*w.d* format 118
$N8601EA*w*. format 120
$N8601EH*w*. format 120
$N8601E*w*. format 119
$N8601EX*w*. format 121
$N8601R*w*. informat 122–123
National Language Support (NLS) 111, 212
NENGO*w*. format 227
NLD language prefix 214
NLDATE. format 235
NLDATE() function 221–225
NLDATEL. format 215, 235
NLDATEM. format 235
NLDATEMD. format 215, 235
NLDATEMDL. format 215, 235
NLDATEMDM. format 216, 235
NLDATEMDS. format 216, 235
NLDATEMDT. format 217
NLDATEML. format 217
NLDATEMN. format 216, 235
NLDATES. format 216, 235
NLDATE*w*. format 215
NLDATEW. format 235
NLDATEWN. format 216, 235
NLDATEW*w*. format 216
NLDATEYM. format 216, 235
NLDATEYML. format 216, 235
NLDATEYMM. format 216, 235
NLDATEYMS. format 216, 235
NLDATEYQ. format 216, 235
NLDATEYQL. format 217, 236
NLDATEYQM. format 217, 236
NLDATEYQS. format 217, 236
NLDATEYR. format 217, 236
NLDATEYW. format 217, 236
NLDATM() function 221–225
NLDATMAP*w*. format 217
NLDATM*w*. format 215, 217
NLS (National Language Support) 111, 212
"NLS" formats 214–220
"NLS" informats 220
NLTIME. format 236
NLTIME() function 221–225
NLTIMEAP. format 236
NODATE system option 7
NOR language prefix 214
numeric variables
  date, time as 3–5
  functions from 140–145
  INPUT() function and 61–62
  LENGTH statement and 5–6
NWKDOM() function 144, 170

O

%o date directive 223
OBSERVED= option, CONVERT statement (EXPAND) 206–212
OBSERVED=AVERAGE option, CONVERT statement (EXPAND) 207–212
OBSERVED=BEGINNING option, CONVERT statement (EXPAND) 207–212
OBSERVED=DERIVATIVE option, CONVERT statement (EXPAND) 207–212
OBSERVED=END option, CONVERT statement (EXPAND) 207–212
OBSERVED=MIDDLE option, CONVERT statement (EXPAND) 207–212
OBSERVED=TOTAL option, CONVERT statement (EXPAND) 207–212
ODS destinations 13–14, 214–215
OPTIONS INTERVALDS= statement 169–176
OPTIONS statement
  DATE/NODATE system option 7
  LOCALE= system option 212
OS TIME macro 76, 78
OUTLIB= option 52–55

P

%p date directive 49, 223
PDF destination 14
PDJULG. informat 66
PDJULG4. informat 66
PDJULG*w*. format 22
PDJULI1. format 23
PDJULI*w*. format 22–23
PDJULI*w*. informat 23, 66
PDTIME4. informat 76
period (.)
 format syntax and 16
 in informats 59
 missing values and 50
PICTURE statement, FORMAT procedure 47–52, 246
POL language prefix 214
PRINT procedure 11–12
PROC step 11
PTG language prefix 214
PUT() function 14, 17, 28, 55–56, 61–62, 221, 244–245
PUTN() function 14, 17, 28, 55–56

Q

%QSYSFUNC() macro function 187
QTR() function 138
QTR interval 150
 INTNX() function 158
 shift point 160
QTRR*w*. format 23–24, 232
QTR*w*. format 23, 232

R

&RAWDATE macro variable 187
retail calendar intervals 181–183
RMFDUR4. informat 76
RMFSTAMP8. informat 80
RTF destination 14
RUS language prefix 214

S

%S date directive 49, 223
sampling frequency
 converting to higher 202–203
 converting to lower 203–205
 OBSERVED= option and 207–212
SASDATEFMT= system option 228–229
SAS/ETS 183, 200
SAS/GRAPH 194–200
seasonality functions 183
SECOND() function 140
SECOND interval 151
 INTNX() function 158
 shift point 161
SEMIMONTH interval 150
 INTNX() function 158
 shift point 160
SEMIYEAR interval 150
 INTNX() function 158
 shift point 160
shift operators, intervals and 159–162
SLO language prefix 214
SMFSTAMP8. informat 80
SMRSTAMP8. informat 80
SQL procedure 193
STIMER system option 76
STIMER*w*. informat 76–77
STRIP() function 14
SVE language prefix 214
&SYSDATE automatic macro variable 185, 187
&SYSDATE9 automatic macro variable 186, 187
&SYSDAY automatic macro variable 186, 187
%SYSFUNC() macro function 187–189
&SYSTIME automatic macro variable 186, 187

T

Taiwanese date formats 226
Taiwanese date informats 226–227
TENDAY interval 150
 INTNX() function 158
 shift point 160
time
 automatic macro variables 186–189
 CALL SYMPUT() function and 192–193
 as constants 3–5
 counters for 2
 custom formats 47–55

default justification 13
Excel and 227–228
external representation of 2–3
formats for 37–41, 92–103
informats for 73–78, 81–82, 103–111
internal representation of 2
international formats and informats 212–220
interval definitions 150–151
interval functions 151–159
quick reference 233, 235–237
shifting intervals 159–168
%SYSFUNC() macro function and 187–189
width specification 5–6, 37
TIME() function 138
time zone functions 111–115
TIMEAMPM11. format 11–13
TIMEAMPM*w.d* format 40
clock values and 37
quick reference 233
TIMEPART() function 41–42, 140
troubleshooting 242–243
TIME*w.* informat 77–78
ANDTDTE*w.* informat and 82
ANYDTDTM*w.* informat and 84
ANYDTTME*w.* informat and 85–86
TIME*w.d* format 39–40
HHMM*w.d* format and 38
quick reference 233
TIMEZONE= option 93, 111–112
TITLE statement 186
titles, date in 186–187
TO= option, EXPAND procedure 206, 209–212
TODAY() function 137
TODSTAMP8. informat 78
TOD*w.d* format 40–41
clock values in 37
quick reference 233
troubleshooting dates 239–251
TU4. informat 78
two-digit year
extraction functions and 138–140
YEARCUTOFF= system options and 57–59, 79
TZONEDSTNAME() function 114
TZONEDSTOFF() function 114
TZONEID() function 112
TZONENAME() function 112–113
TZONEOFF() function 113, 246
TZONES2U() function 113
TZONESTTNAME() function 114–115
TZONESTTOFF() function 115
TZONEU2S() function 115, 246

U

U algorithm 25, 35, 69, 70, 139
%u date directive 223
%U date directive 49, 223

V

V algorithm 26–27, 36–37, 69–71, 139, 223
%V date directive 223
VALUE statement, FORMAT procedure 47–48
variables
See character variables; macro variables; numeric variables

W

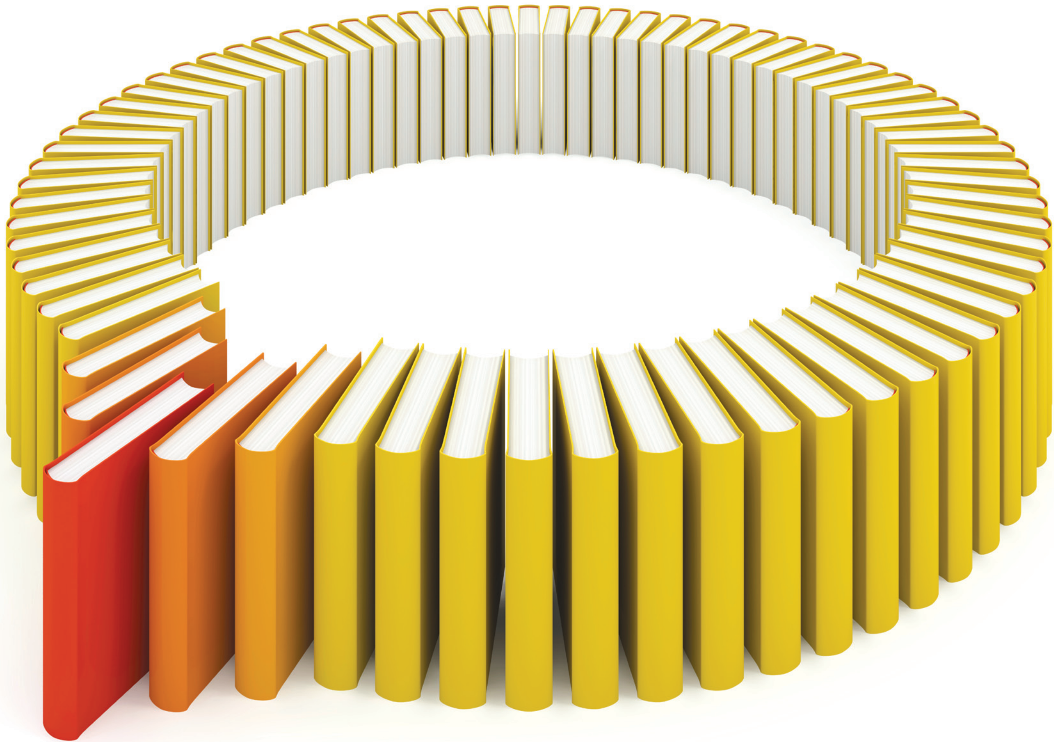
W algorithm 27, 37, 69–72, 139, 223
%w date directive 49, 223
%W date directive 223
WEEK() function 139
WEEK informats 69–70
WEEK interval 150
INTCK() function 152
INTNX() function 157, 159
shift point 160, 161–162, 163–164
WEEKDATE*w.* format 11–12, 24
quick reference 232
WEEKDATX*w.* format and 24–25
WEEKDATX*w.* format 24–25
DTWKDATX*w.* format and 445
international format for 213–214
quick reference 232
WEEKDAY() function 139
WEEKDAY interval 150
INTNX() function 157
shift point 160
WEEKDAY*w.* format 25
international format for 213

quick reference 232
WEEKU*w.* format 25–26
quick reference 232
WEEKU*w.* informat 70–71
WEEKV interval 182–183
WEEKV*w.* format 26–27
quick reference 232
WEEKW*w.* format and 27
WEEKV*w.* informat 71–72
WEEKW*w.* format 27, 232
WEEKW*w.* informat 72–73
WHERE clause, SQL procedure 228
width specification
for date formats 5–6, 13–14
for datetime formats 5–6, 37, 41–42
for time formats 5–6, 37
formats and 13–14
informats and 59
WORDDATE*w.* format 28
FORMAT statement and 50
quick reference 232
WORDDATX*w.* format 28–29, 249
international format for 213
quick reference 232
WORKINGDAYS interval 175–176

Y

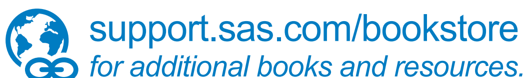
%y date directive 49, 223
%Y date directive 49, 51, 223
Y2K problem 185
YEAR() function 139
YEAR interval 150
INTCK() function 152
INTNX() function 158
shift point 160, 161–162, 165–166
YEAR10. interval 166–167
YEARCUTOFF= system option
DATJUL() function and 140–141
extraction functions and 138–140
Japanese/Taiwanese date informats 226–227
MDY() function 143
PDJULG*w.* format and 22
PDJULI. format 23
two-digit year and 57–59, 79
YYQ() function 144–145
YEARV interval 182
YEAR*w.* format 29
DTYEAR*w.* format and 45
quick reference 232
YMDDTTM*w.d* informat 80–81
YRDIF() function 146–147, 155–159
YYMM. format 232
YYMMC. format 30, 232
YYMMD. format 30, 232
YYMMDDB. format 32, 232
YYMMDDC. format 32, 232
YYMMDDD. format 32, 232
YYMMDDN. format 32, 232
YYMMDDP. format 32, 232
YYMMDDS. format 232
YYMMDD*w.* format 30–31
DATESTYLE= system option 81–82
quick reference 232
YYMMDD*w.* informat 66–67
ANYDTDTE*w.* informat and 82
ANYDTDTM*w.* informat and 84
ANYDTTME*w.* informat and 85–86
DATESTYLE= system option and 81–82
YYMMDD*xw.* format 31–32
YYMM*xw.* format and 30
YYQ*xw.* format and 33–34
YYMMN. format 30, 232
YYMMN*w.* informat 67
YYMMP. format 30, 232
YYMMS. format 232
YYMM*w.* format 29
YYMM*xw.* format 30
YYMON*w.* format 32, 232
YYQ() function 144–145
YYQC. format 33
DDTYQC*w.* format and 46
quick reference 233
YYQD. format 34, 233
YYQN. format 33, 233
YYQP. format 34, 233
YYQRC. format 35, 233
YYQRD. format 35, 233
YYQRN. format 35, 233
YYQRP. format 35, 233
YYQRS. format 233

YYQR*w*. format 34, 233
YYQR*xw*. format 34–35
YYQS. format 233
YYQ*w*. format 32–33, 233
YYQ*w*. informat 67–68
  ANYDTDTE*w*. informat and 82
  ANYDTDTM*w*. informat and 84
  ANYDTTME*w*. informat and 85–86
YYQ*xw*. format 33–34
YYWEEKU. format 233
YYWEEKU*w*. format 35–36
YYWEEKV. format 233
YYWEEKV*w*. format 36
YYWEEKW*w*. format 37, 233



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.



SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0413