QNX SOFTWARE SYSTEMS

Implementing Device Drivers

A Case Study in Migrating from Linux to a Microkernel OS

by David Donohoe, Senior Software Developer
QNX Software Systems Ltd.
ddonohoe@qnx.com

Abstract

Recent market developments around IP ownership have prompted many companies to investigate the possibility of migrating their Linux code to a commercial-grade operating system. This paper describes a case study that examines the scope of the effort involved in migrating device driver code developed for a “monolithic” OS such as Linux to a microkernel OS such as the QNX® Neutrino® realtime OS (RTOS). A primary goal of the study was to provide an indication of how much, if any, of an existing code base can be leveraged during this type of migration.

This paper will be of interest to developers and development managers who are considering deploying their products on a commercial RTOS, and who have an existing code base written on Linux. The intention of the study is to provide an indication of the effort involved in such a migration, and how much of the existing device driver code can be re-used.

There were two selection criteria for the case driver: that it be unsupported under QNX Neutrino, and that it have readily available source code. This paper summarizes the steps involved in the migration effort, the differences — and similarities — between the two OS environments from a driver implementation perspective, and the results of the code migration.

Challenges of Migrating to a Microkernel-based Operating System

At the application level, migrating from Linux to QNX Neutrino is usually quite straightforward, even though they are very different systems in terms of implementation: Linux is based on a more traditional "monolithic" kernel design, whereas QNX Neutrino is based on a microkernel design. The reason for this ease of migration is that both OSs are designed to be compatible with POSIX standards; hence applications coded to be POSIX compatible can be easily migrated from one type of system to the other. Often, migrating such an application is simply a case of recompiling the source code, with no modifications to the source code being necessary.[1]

Unfortunately, this isn't the case when it comes to device-driver code. This is in part because of the differences between the microkernel and monolithic kernel implementations. With a microkernel, device driver functionality isn't part of the system kernel. Rather, devices are typically accessed and managed via a set of separate co-processes. There isn't really any difference between a device management co-process and a regular application process. In fact, a device management process has the same POSIX API available to it as a regular application.

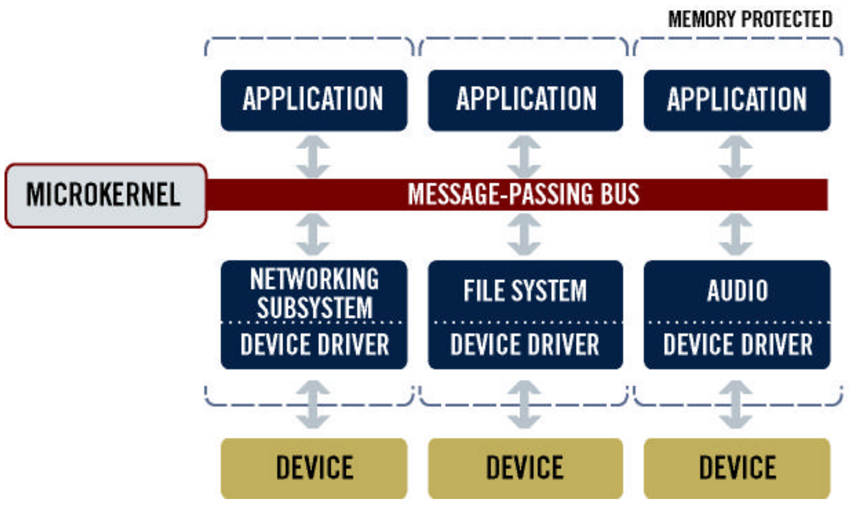

Figure 1 – QNX Neutrino Microkernel RTOS Architecture

[1] For examples of open source applications that can be readily ported to QNX Neutrino, download the white paper "Real Time or Real Linux? A Realistic Alternative" from http://www.qnx.com/mailings/web_app/.

Contrast this to a monolithic kernel implementation, where device management code resides *inside* the kernel. When executing within the kernel, it isn't possible to call operating system functions in the normal way; therefore the facilities of the POSIX API aren't available to device drivers. Naturally, this has a major impact on how device drivers will be written.

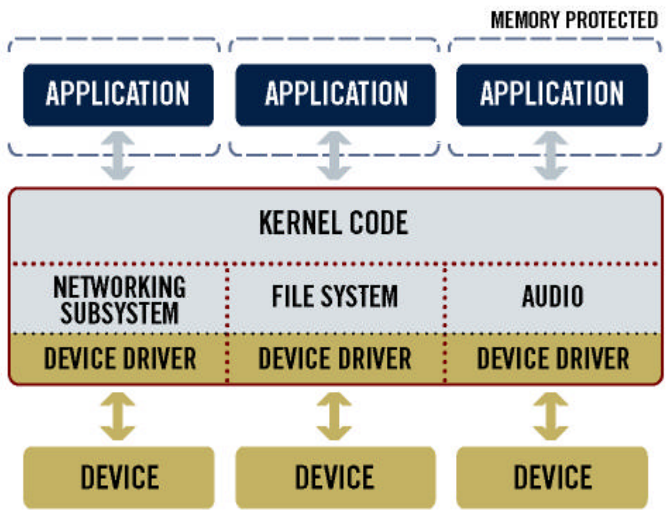

Figure 2 – Monolithic OS Architecture

Still, it isn't necessarily this difference in OS architecture that makes driver migration from Linux to QNX Neutrino non-trivial. In fact, it could be said that migrating from one monolithic OS to another poses just as much of a challenge. This is because device drivers are generally written to a specification that is proprietary to a single operating system. The reality is that device drivers are rarely written to a universal specification that allows them to be compatible with more than one OS. There is no standard for device driver implementation that is as widely adopted as POSIX is for application implementation. Device driver code can therefore take a significant amount of reworking before it can "plug in" to an operating system environment other than that for which it was targeted.

Notwithstanding the fundamental differences in system architecture, both monolithic and microkernel implementations have proprietary interfaces between device drivers and the rest of the system.

Choosing a Candidate Driver for the Case Study

For the purposes of this case study, we had to select a candidate driver written for a monolithic kernel design, for porting to QNX Neutrino. Since Ethernet devices are one of my main areas of expertise, I decided to look for an Ethernet driver that was readily available in source code form. (Obviously, we are interested in studying the effort involved for someone with experience in a given type of device driver.) The driver for the SMC9452TX Gigabit Ethernet PCI adapter ended up being chosen, because:

- The driver was available, in source code format, on SMC's website.
- The SMC 9452TX wasn't already supported under QNX Neutrino.

The source code to the SMC driver is published under the GNU Public License (GPL). Since this case study is merely an academic exercise, in that the ported code isn't intended to be made part of any product, the implications of the GPL are, in this example, not an issue.

Porting the SMC 9452TX Linux Driver Source to QNX Neutrino

The SMC 9452 driver source was well-structured and well-written, and therefore lent itself to a relatively straightforward porting effort. Structurally, the driver was divided into two parts: a hardware-specific portion and an OS-specific portion. The hardware specific portion was approximately 7400 lines of code, whereas the OS specific portion was 2030 lines of code.

While the interfaces that a Linux and a QNX device driver use are different, the effective functionality that the interfaces provide are quite similar. Browsing through the source code, I identified the main interactions between the driver and the rest of the system, to try to determine the scope of the code modifications that would be necessary.

Low-level interaction with hardware

Both operating systems provide straightforward mechanisms whereby a driver can access memory-mapped or I/O-mapped device registers.

Device detection and configuration

Both operating systems provide mechanisms to automatically configure devices on the PCI bus, including library calls that allow the driver to detect the presence of a device and determine the device's configuration. Both operating systems also provide mechanisms that let the user supply configuration parameters that affect the driver's operation. These parameters are implemented quite differently under the two operating systems. Under QNX Neutrino, device driver options are parsed, and the configuration information is stored in private per-device data structures. Under Linux, after the options are parsed, the information is stored in global variables within the driver module.

Interrupt handling

Both operating systems provide a mechanism to allow a thread to execute the driver's interrupt-handling code in response to interrupt-generating events triggered by the device.

Network traffic — sending and receiving packets

The format of packet data buffers, and the API for transferring data between the device driver and the operating system's networking infrastructure, are quite different, but the principle is the same.

Miscellaneous networking functionality

This includes functionality such as multicast address filtering and statistics gathering. Again, while the APIs for providing the functionality differ, the principles are the same.

Hardware-specific Code

A clear picture begins to emerge with respect to what work needs to be done. The overall structure of the driver can be maintained, but portions of the operating system-specific code need to be rewritten. In some cases, significant portions of code need to be implemented or re-implemented; for example, command-line option string parsing, which is a unique aspect of QNX drivers. Conveniently, such portions of code already exist: there are already many PCI network adapters supported under QNX Neutrino, such as the "pcnet" driver supplied with the QNX Neutrino v6.x network driver development kit.

It turns out that the hardware-specific piece of code (all 7400 lines of it) didn't even need to be edited. After all, it was just standard ANSI C code that communicates with the SMC 9452 via a handful of macros from a header file. These macros were reimplemented for QNX in the header file, and ended up as follows:

```
#define NsmRegRead8(ctx, addr, pval)     *(pval) = (*(volatile uint8_t *)(addr))
#define NsmRegRead16(ctx, addr, pval)    *(pval) = (*(volatile uint16_t *)(addr))
#define NsmRegRead32(ctx, addr, pval)    *(pval) = (*(volatile uint32_t *)(addr))

#define NsmRegWrite8(ctx, addr, val)     *(volatile uint8_t *)(addr) = (val)
#define NsmRegWrite16(ctx, addr, val)    *(volatile uint16_t *)(addr) = (val)
#define NsmRegWrite32(ctx, addr, val)    *(volatile uint32_t *)(addr) = (val)
```

Since the hardware-specific code is passed a pointer to the device's memory-mapped registers, this is all that it needed to communicate with the SMC 9452 hardware.

OS-specific Code

Next comes the hard part — the OS-specific portion. Since I was able to maintain the overall structure of the OS-specific portion, I ended up with basically the same set of functions, but with mostly different code. The original Linux code was used as a reference for how the OS-specific code needed to interact with the hardware-specific code. I was able to leverage a lot of code from a previously developed QNX Neutrino driver for a different PCI network adapter, which greatly reduced the development time. In the end, having replaced most of the OS-specific code, I had about 1600 lines of code in the new QNX Neutrino driver, in place of the 2030 lines of code from the original Linux driver. Overall, about 80 percent of the code from existing Linux driver was preserved.

Results

After about 10 hours of development, I had a driver that provided the basic functionality expected of an Ethernet driver:

- packet transmission and reception
- statistics gathering
- multicast address filtering

Additionally, the driver followed the normal QNX Neutrino driver conventions for option parsing, link detection/selection, and so on.

Performance Testing

I measured the performance of the resulting driver, using the "netperf" TCP throughput measuring utility. It should be noted that, since this was a case study exercise only, there was no work done to optimize the driver performance.

The machine configurations were as follows:

- **dinah** — 2GHz. Pentium IV, with an Intel 82544 Gigabit adapter.
- **moe** — 450Mhz. Pentium III, with an SMC 9452TX Gigabit adapter.

Netperf test - Intel 82544 to SMC 9452

```
dinah:/home/ddonohoe>netperf -fK -p8000 -Hmoe
TCP STREAM TEST to moe
```

Recv socket size (bytes)	Send socket size (bytes)	Send message size (bytes)	Elapsed time (secs)	Throughput (kbytes/sec)
16384	22528	22528	10.00	28706.91

Netperf test - SMC 9452 to Intel 82544

```
moe:/home/ddonohoe>netperf -fK -p8000 -Hdinah
TCP STREAM TEST to Dinah
```

Recv socket size (bytes)	Send socket size (bytes)	Send message size (bytes)	Elapsed time (secs)	Throughput (kbytes/sec)
16384	22528	22528	10.00	20661.70

Conclusion

It becomes apparent that having an existing code base of driver level software for another OS can greatly accelerate migration to a different OS, even if the two OSs have fundamental architectural differences. In this case, we were able to leverage about 80 percent of the original driver code, allowing an initial driver port to be completed in a very short period of time. Thus, having an existing base of driver code available, as opposed to having to implement device drivers from scratch, dramatically reduces the development efforts required to migrate.

As an aside, the fast development time could be partially attributed to the memory-protected microkernel architecture. During development, I made some mistakes that caused my code to crash. However, I never once had to reboot the system: I simply had to fix the offending sections of code and restart the networking subsystem with the updated driver. To anyone who has previously developed drivers for a monolithic kernel, where a driver bug will crash the entire operating system, the debugging benefits of this rapid crash recovery are very obvious.

Naturally, the more the developers doing the migration know about the technologies related to the drivers they are porting, the better equipped they will be for the task. In the case of Ethernet, for example, anyone who is qualified to write an Ethernet driver for a multi-tasking, memory-protected OS should be able to do a driver port in a relatively short timeframe. That said, it should be noted that the more familiar the developers are with programming for the target OS, the more quickly they will be able to complete the porting work.

When developing drivers for QNX Neutrino, a background in POSIX application development can be beneficial, since the QNX device drivers have the POSIX API available to them.

About QNX Software Systems

Founded in 1980, QNX Software Systems is the industry leader in realtime, microkernel OS technology. The inherent reliability, scalable architecture, and proven performance of the QNX Neutrino RTOS make it the most trusted foundation for future-ready applications in the networking, automotive, medical, and industrial automation markets. Companies worldwide like Cisco, Ford, Johnson Controls, Siemens, and Texaco depend on the QNX technology for their mission- and life-critical applications. Headquartered in Ottawa, Canada, QNX Software Systems maintains offices in North America, Europe, and Asia, and distributes its products in more than 100 countries worldwide.

www.qnx.com

QNX SOFTWARE SYSTEMS

www.qnx.com