QNX Software Systems Ltd.
175 Terence Matthews Crescent
Ottawa, Ontario, Canada, K2M 1W8
Voice: 1 800 676-0566 +1 613 591-0931
Email: info@qnx.com
Web: www.qnx.com

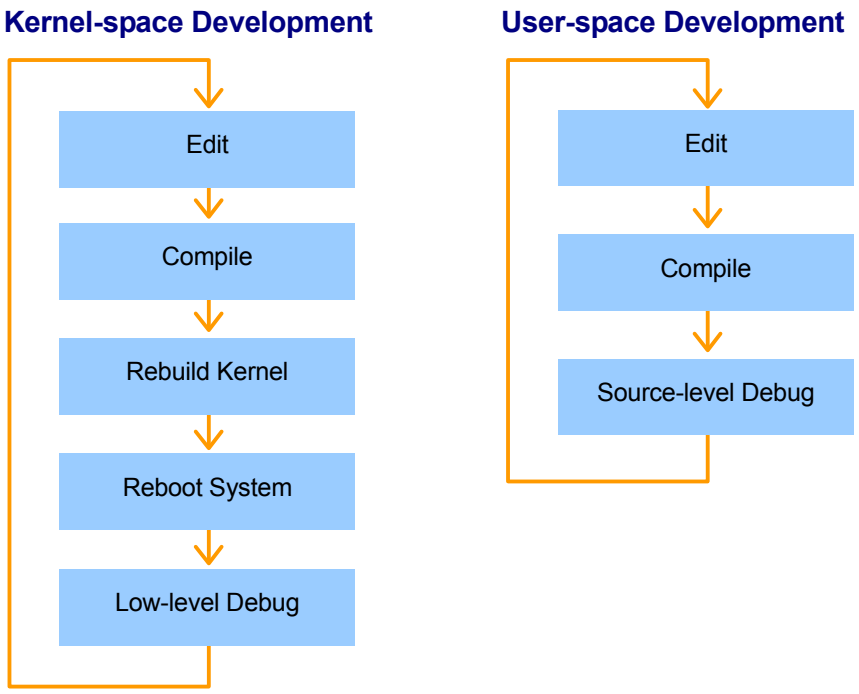
User-Space Debugging Simplifies Driver Development

Chris McKillop
QNX Software Systems Ltd.
cdm@qnx.com

Introduction

The traditional approach to driver development, in which every driver is written and debugged as part of the OS kernel, can seriously hamper the progress of an embedded project. For instance, a single programming error in any kernel-space driver can crash the target system. As a result, driver developers often waste time rebuilding and rebooting the target instead of actually testing and debugging software. To complicate matters, most embedded systems lack the non-volatile storage needed to save a kernel core dump between reboots. This makes post-mortem debugging — which would help locate the source of the system failure — nearly impossible. As a further problem, kernel debuggers typically halt the entire system while the developer inspects the code or data of the driver being debugged. Because everything must run in lockstep with the debugger, the developer can easily miss bugs that would occur in a live system, where events happen asynchronously.

Fortunately, not all OSs rely on in-kernel drivers. The QNX® Neutrino® microkernel OS, for instance, treats every driver as a standard, user-space application. So, if any driver fails, the OS can cleanly terminate the driver and reclaim all the resources it was using; there's no need to rebuild and reboot. Better yet, drivers can be debugged with the same, standard process-level debuggers used to debug regular applications. There's no need to learn separate kernel-debug tools and no need to halt the entire system while a driver is being debugged. All other software processes continue to run normally.



When a kernel-space driver fails, the developer typically has to rebuild and reboot the entire target. But when a user-space driver fails, the developer can simply recompile and download the new driver.

Even if your OS doesn't directly support user-space drivers, it will probably let you develop and debug significant parts of a driver in user space. You can then move the finished driver into the kernel to gain access to system services that don't have user-space interfaces. But whether your driver ultimately runs in kernel space or user space, it must still do the following:

- manipulate hardware registers
- access specific memory locations
- handle interrupts
- interact with other parts of the system

Since these attributes aren't normally associated with processes running in user space, you'll need to take several steps to ensure that the user-space driver operates correctly...

Step 1: Gain Basic Privileges

First, your user-space driver must gain appropriate privileges from the operating environment. On a UNIX/POSIX system, the driver will need to run as `root` (UID 0) and may also need to invoke a system-specific call to gain whatever additional permissions that drivers require. For example, in the QNX Neutrino RTOS, a driver will call *ThreadCtrl*(), which gives the driver full I/O and interrupt privileges. On Linux, an equivalent function is *iopl*().

Step 2: Gain Access to the Device

At this point, the driver has the rights to access hardware and, on some CPU platforms, can start using instructions to access I/O ports or specific addresses in physical memory. In the latter case, the driver must ask the operating environment to set up a mapping between the required physical-address region and the driver's virtual address space. This is required since, as a user-space process, the driver runs in virtual memory. It may also be necessary to disable caching in this physical-address region to ensure that the driver reads and writes directly to the device.

If you're working with a POSIX-compliant operating system, you can use a standard call, *mmap*(), to map address spaces. (Many non-POSIX systems provide an equivalent call.) Most developers with a UNIX background are familiar with *mmap*() as a way to map a file on disk into the address space of a process. But it can also map known, physical addresses into a process's address space, although the manner in which *mmap*() is used to do this isn't part of the POSIX standard. Nonetheless, most POSIX systems follow common conventions, including support for a special device, /dev/mem, which represents the entire physical-address space of the machine. A driver can open /dem/mem in the same way as a file, and then invoke *mmap*() to bring the desired section of physical-address space into the driver's virtual address space. The following example shows code that will "map in" the text-mode address space of a standard VGA card on a standard x86 PC:

Mapping a VGA text buffer

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/mman.h>
```

```
/* VGA Text Mode Segment Details */
#define MY_MAP_ADDR (0xb0000)
#define MY_MAP_SIZE (1024*64)

int main( void )
{
    int fd;
    void *ptr;

    fd = open( "/dev/mem", O_RDWR );
    if( fd == -1 )
        return EXIT_FAILURE;

    ptr = mmap( 0, MY_MAP_SIZE,
                PROT_READ | PROT_WRITE,
                MAP_SHARED,
                fd, MY_MAP_ADDR );

    if( ptr == MAP_FAILED )
        return EXIT_FAILURE;

    /* Do something to the frame buffer */

    munmap( ptr, MY_MAP_SIZE );
    close( fd );
    return EXIT_SUCCESS;
}
```

Step 3: Handle Interrupts

If your device driver doesn't have to deal with interrupts, you can write a production-quality driver in user space regardless of your operating system. Graphics drivers, for instance, can be easily be done from user-space — the XFree86 project represents a good example.

Nonetheless, most drivers must support interrupts. Problem is, most OSs don't provide any facilities to propagate interrupt events from kernel space into user space. In such cases, your user-space driver will simplify initial development and debugging, but the final version will have to run in the kernel to avoid the overhead of user-space interrupt simulation.

User-space interrupt handling, when supported, is very specific to the operating system. For example, QNX Neutrino provides APIs (*InterruptAttach*, *InterruptAttachEvent*) for different interrupt-handling modes. Because there's no standard method, let's look instead at how to simulate user-space interrupts on a system that doesn't support them. This will let you debug the driver in user space, even if it must ultimately run as part of the kernel.

Interrupt handlers are simply callback functions that are invoked asynchronously from the rest of the operating environment when the hardware being driven needs to be serviced or completes an operation. (This process is a very similar to how signal handlers are used by user-space applications to handle asynchronous events.) Since the real interrupt can't be asserted in the driver itself, the driver must poll the hardware to detect events that would normally cause an interrupt to occur.

You can do this by setting up a signal handler and creating a timer. When the timer "times out," the signal handler will be invoked. Within the signal handler, the hardware will be polled to see if a real interrupt has occurred or if the conditions that would cause a real interrupt have been met. Good starting values for polling will range from 10 to 100 milliseconds. This is just an estimate, however: if you set the frequency too high the system will become overloaded, and if you set it too low you could miss events and not get realistic performance numbers out of your driver. The following code example shows how to set up this simulation on a POSIX system.

Simulating an interrupt by polling

```
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <time.h>

static int PollCount = 0;

void interrupt_handler( int signo )
{
    /* Poll hardware and check for interrupt. Process if found. */
    printf( "Polling Hardware: %d\n", PollCount++ );
}

int main( void )
{
    int ret;
    timer_t timerid;
    struct sigevent sigev;
    struct itimerspec itime;

    /* Install the virtual interrupt handler */
    signal( SIGUSR1, interrupt_handler );

    /* Create the timer, and have it raise SIGUSR1 when it expires */
    memset( &sigev, 0, sizeof( sigev ) );
    sigev.sigev_notify = SIGEV_SIGNAL;
    sigev.sigev_signo = SIGUSR1;

    ret = timer_create( CLOCK_REALTIME, &sigev, &timerid );
    if( ret < 0 )
        return EXIT_FAILURE;
```

```
    /* Set the timer's timeout value, 100ms */
    memset( &itime, 0, sizeof( itime ) );
    itime.it_value.tv_sec = 0;
    itime.it_value.tv_nsec = 100000000;
    memcpy( &itime.it_interval, &itime.it_value, sizeof( itime.it_value ) );

    ret = timer_settime( timerid, 0, &itime, NULL );
    if( ret < 0 )
        return EXIT_FAILURE;

    /* Normally would be blocking for requests, simulate by simply blocking */
    while( 1 )
        sleep( 10 );

    return EXIT_SUCCESS;
}
```

Step 4: Handle System Interactions

When writing kernel-space drivers, you have to ask the kernel to handle a special device file on the driver's behalf; this file is normally found under /dev. When I/O operations occur on the special file, generally via *ioctl*()calls, the kernel will route the data and requests to the driver and from the driver to the application. In OSs such as QNX Neutrino, where user-space drivers are the norm, the OS will provide an equivalent mechanism to route messages to and from the driver process. But in OSs where user-space drivers aren't the norm, the user-space driver must rely on an existing form of interprocess communication (IPC) provided by the operating system and wrap a custom API on the transport.

For instance, the driver could use sockets (TCP or UNIX), SystemV IPC, named pipes (FIFOs), or shared memory. The first method, sockets, offers two notable benefits: easy-to-use bidirectional communication and a standard, cross-platform API. Although the exact details of writing a client-server socket application extend beyond the scope of this article, the basic flow is simple:

1) When the user-space driver starts, it creates a socket, binds that socket to a known address, and then waits for requests to service;

2) When an application needs to interact with the driver, it simply opens a connection to the driver's socket and begins to pass data back and forth over the connection. You can achieve the same results using POSIX or SystemV message queues or, for that matter, any other form of IPC provided by the operating environment.

Using DDKs to Speed Driver Development

An OS may provide additional facilities to simplify development of user-space drivers. The QNX Neutrino RTOS, for instance, supports off-the-shelf driver development kits (DDKs) for a variety of device types, including audio, character, disk, graphics, input, networking, parallel, printer, serial, and USB. The kits include detailed documentation, ready-to-customize source code, and a framework that implements all higher-level, device-independent code in libraries — the only code you have to write is the hardware-specific code for your device. The DDKs are available in the QNX Momentics® development suite, which provides tightly integrated, graphical tools (e.g. debugger, profiler, memory analysis tool, system analysis tool) to help you debug and optimize driver code. For information on QNX Momentics, visit http://www.qnx.com/products/ps_momentics/

If you wish to target multiple processor families, QNX Neutrino also provides functions and macros that let you write processor-independent drivers and applications. In some cases, you simply need to specify a new processor target to generate binaries for a different processor — you can even build code for multiple processor families simultaneously. QNX Neutrino supports a wide variety of processors, including ARM, MIPS, PowerPC, SH-4, StrongARM, XScale, and x86.

Building Self-Healing Systems

Writing drivers in user-space isn't difficult. In fact, it has many benefits, even if you can use the user-space version only for initial development. Nonetheless, if your OS allows fully functional user-space drivers to run in your final product, as QNX Neutrino does, you can achieve an additional benefit: much greater reliability. This reliability comes from the ease with which faulty drivers can be restarted automatically, without operator intervention and without system resets. Systems can, in effect, heal themselves of driver failures. For many embedded systems this isn't just a desirable feature; it's an essential requirement: telecommunication service providers, for example, now demand virtually 100% uptime, and surgeons can't reboot their medical monitoring equipment in the middle of an operation! Better yet, user-space drivers can be replaced dynamically with new versions, allowing a system to provide continuous service even while being upgraded with new functionality.