Talking to hardware under QNX Neutrino

by Dave Donohoe

If you've ever tried to develop a device driver under a traditional UNIX operating system, you're sure to feel spoiled when developing hardware-level code for QNX Neutrino.

Thanks to Neutrino's microkernel architecture, writing device drivers is like writing any other program. Only core OS services reside in "kernel" address space -- everything else, including device drivers, reside in "process" or "user" address space. The impact of this is that a device driver has all the services that are available to "regular" applications.

Many models are available to driver developers under Neutrino. Generally, the type of driver you're writing will determine the driver model you'll follow. For example, graphics drivers follow one particular model, which allows them to plug into the Photon graphics subsystem, whereas network drivers follow a different model, and so on.

On the other hand, depending on the type of device you're targeting, it may not make sense to follow any existing driver model at all.

In this article, we'll focus on the low-level details of accessing and controlling device-level hardware, which are common to all types of device drivers.

Probing the hardware

If you're targeting a "closed" embedded system with a fixed set of hardware, your driver may be able to assume that the hardware it's going to control is present in the system and is configured in a certain way.

But if you're targeting more generic systems, such as the desktop PC, you want to first determine whether the device is present. Then you need to figure out how the device is configured (e.g. what memory ranges and interrupt level belong to the device).

For some devices, there's a standard mechanism for determining configuration. Devices that interface to the PCI bus have such a mechanism. Each PCI device has a unique "vendor" and "device" ID assigned to it.

The following piece of code demonstrates how, for a given PCI device, to determine whether the device is present in the system and what resources have been assigned to it:

```
#include <stdio.h>
#include <stdlib.h>
#include <hw/pci.h>
main()
{
struct pci_dev_info info;
void *hdl;
int i;
memset(&info, 0, sizeof (info));
if (pci_attach(0) < 0) {
perror("pci_attach");
exit(EXIT_FAILURE);
}
```

```
/*
* Fill in the Vendor and Device ID for a 3dfx VooDoo3
* graphics adapter.
*/
info.VendorId = 0x121a;
info.DeviceId = 5;
if ((hdl = pci_attach_device(0,
PCI_SHARE|PCI_INIT_ALL, 0, &info)) == 0) {
perror("pci_attach_device");
exit(EXIT_FAILURE);
}
for (i = 0; i < 6; i++) {
if (info.BaseAddressSize[i] > 0)
printf("Aperture %d: "
"Base 0x%llx Length %d bytes Type %s\n", i,
PCI_IS_MEM(info.CpuBaseAddress[i]) ?
PCI_MEM_ADDR(info.CpuBaseAddress[i]) :
PCI_IO_ADDR(info.CpuBaseAddress[i]),
info.BaseAddressSize[i],
PCI_IS_MEM(info.CpuBaseAddress[i]) ? "MEM" : "IO");
}
printf("IRQ 0x%x\n", info.Irq);
pci_detach_device(hdl);
}
```

Different buses have different mechanisms for determining which resources have been assigned to the device. On some buses, such as the ISA bus, there's no such mechanism. How do you determine whether an ISA device is present in the system and how it's configured? The answer is card-dependent (with the exception of "PnP" ISA devices).

Accessing the hardware

Once you've determined what resources have been assigned to the device, you're now ready to start communicating with the hardware. How you do this depends on the resources.

1. I/O resources

Before a thread may attempt any port I/O operations, it must be running at the correct privilege level. The following call will ensure that the thread is permitted to access I/O ports:

```
ThreadCtl(_NTO_TCTL_IO, 0);
```

Without this call, you'll get a protection fault upon attempted I/O operations.

Next you need to map the I/O base address (one of the addresses returned in the **CpuBaseAddress** array of the info structure above). For example:

```
uintptr_t iobase;
iobase = mmap_device_io(info.BaseAddressSize[2],
info.CpuBaseAddress[2]);
```

Now you may perform port I/O, using functions such as **in8**(), **in32**(), **out8**(), etc, adding the register index to **iobase** to address a specific register:

```
out32(iobase + SHUTDOWN_REGISTER, 0xdeadbeef);
```

Note that the call to **mmap_device_io()** isn't necessary on x86 systems, but it's still a good idea to include it for the sake portability. In the case of some legacy x86 hardware, it may not make sense to call **mmap_device_io()**. For example, a VGA-compatible device has I/O ports at well-known, fixed locations (e.g. 0x3c0, 0x3d4, 0x3d5) with no concept of an I/O base as such. You could access the VGA controller, for example, as follows:

```
out8(0x3d4, 0x11);
out8(0x3d5, in8(0x3d5) & ~0x80);
```

2. Memory-mapped resources

For some devices, registers are accessed via regular memory operations. To gain access to a device's registers, you need to map them to a pointer in the driver's virtual address space. This can be done by calling **mmap_device_memory()**.

```
volatile uint32_t *regbase; /* device has 32-bit registers */
regbase = mmap_device_memory(NULL, info.BaseAddressSize[0],
PROT_READ|PROT_WRITE|PROT_NOCACHE, 0,
info.CpuBaseAddress[0]);
```

Note that we specified the **PROT_NOCACHE** flag. This ensures that the CPU won't defer or omit read/write cycles to the device's registers, nor will it deliver the reads or writes to the registers in a different order than the driver issued them.

Note also the use of the volatile keyword. This prevents the compiler from "optimizing out" accesses to the devices registers.

Now you may access the device's memory using the **regbase** pointer. For example:

```
regbase[SHUTDOWN_REGISTER] = 0xdeadbeef;
```

3. IRQs

You can attach an interrupt handler to the device by calling either **InterruptAttach**() or **InterruptAttachEvent**(). For example:

```
InterruptAttach(_NTO_INTR_CLASS_EXTERNAL | info.Irq,
handler, NULL, 0, _NTO_INTR_FLAGS_END);
```

The driver should call **ThreadCtl(_NTO_TCTL_IO, 0)**; before attaching an interrupt.

The essential difference between **InterruptAttach()** and **InterruptAttachEvent()** is the way in which the driver is notified that the device has triggered an interrupt.

With **InterruptAttach()**, the driver's "handler" function is called directly by the kernel. Since it's running in kernel space, the hander is severely restricted in what it can do. From within this handler, it isn't safe to call most of the C library functions. Also, if you spend too much time in the handler, other processes and

interrupt handlers of a lower or equal priority won't be able to run. Doing too much work in the interrupt handler can negatively affect the system's real-time responsiveness.

We recommend that you do the bare minimum within the handler and return an event to be delivered to the driver at process level. Then, the rest of the work associated with handling the interrupt can be completed at process time and will be carried out at the driver's normal priority.

Typically, a driver would simply acknowledge the interrupt at the hardware level within its interrupt handler and would return an event in order to wake up the driver thread that will perform the remainder of the processing.

It's often possible to do all the interrupt handling at the process level. In this case, you should call **InterruptAttachEvent()**. When the driver triggers an interrupt, the kernel will automatically deliver an event to the driver.

Before attempting to implement an interrupt handler, you should read the online documentation very carefully. For more information, see "Writing an Interrupt Handler" in the QNX Neutrino Programmer's Guide.

Now you should be ready to start programming the device's registers.