BRIEF CONTENTS

About the Author and the Technical Reviewer .............................................................. xvii

Foreword by John Baldwin .......................................................................................... xix

Acknowledgments ...................................................................................................... xxii

Introduction ............................................................................................................... xxiii

Chapter 1: Building and Running Modules .............................................................. 1

Chapter 2: Allocating Memory ............................................................................... 17

Chapter 3: Device Communication and Control ..................................................... 27

Chapter 4: Thread Synchronization ...................................................................... 53

Chapter 5: Delaying Execution ............................................................................ 83

Chapter 6: Case Study: Virtual Null Modem ......................................................... 99

Chapter 7: Newbus and Resource Allocation .......................................................... 113

Chapter 8: Interrupt Handling ............................................................................. 125

Chapter 9: Case Study: Parallel Port Printer Driver .............................................. 141

Chapter 10: Managing and Using Resources ......................................................... 165

Chapter 11: Case Study: Intelligent Platform Management Interface Driver ....... 183

Chapter 12: Direct Memory Access ..................................................................... 193
Chapter 13: Storage Drivers ................................................................. 207
Chapter 14: Common Access Method .................................................. 225
Chapter 15: USB Drivers .................................................................. 257
Chapter 16: Network Drivers, Part 1: Data Structures ......................... 283
Chapter 17: Network Drivers, Part 2: Packet Reception and Transmission ......................................................... 299
References ....................................................................................... 309
Index .................................................................................................. 311
NEWBUS AND RESOURCE ALLOCATION

Until now, we’ve examined only pseudo-devices, which provide a superb introduction to driver writing. However, most drivers need to interact with real hardware. This chapter shows you how to write drivers that do just that.

I’ll start by introducing Newbus, which is the infrastructure used in FreeBSD to manage the hardware devices on the system (McKusick and Neville-Neil, 2005). I’ll then describe the basics of a Newbus driver, and I’ll conclude this chapter by talking about hardware resource allocation.

Autoconfiguration and Newbus Drivers

Autoconfiguration is the procedure carried out by FreeBSD to enable the hardware devices on a machine (McKusick and Neville-Neil, 2005). It works by systematically probing a machine’s I/O buses in order to identify their child...
devices. For each identified device, an appropriate Newbus driver is assigned to configure and initialize it. Note that it’s possible for a device to be unidentifiable or unsupported. As a result, no Newbus driver will be assigned.

A Newbus driver is any driver in FreeBSD that controls a device that is bound to an I/O bus (that is, roughly every driver that is not a pseudo-device driver).

In general, three components are common to all Newbus drivers:

- The device_foo functions
- A device method table
- A DRIVER_MODULE macro call

**device_foo Functions**

The device_foo functions are, more or less, the operations executed by a Newbus driver during autoconfiguration. Table 7-1 briefly introduces each device_foo function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device_identify</td>
<td>Add new device to I/O bus</td>
</tr>
<tr>
<td>device_probe</td>
<td>Probe for specific device(s)</td>
</tr>
<tr>
<td>device_attach</td>
<td>Attach to device</td>
</tr>
<tr>
<td>device_detach</td>
<td>Detach from device</td>
</tr>
<tr>
<td>device_shutdown</td>
<td>Shut down device</td>
</tr>
<tr>
<td>device_suspend</td>
<td>Device suspend requested</td>
</tr>
<tr>
<td>device_resume</td>
<td>Resume has occurred</td>
</tr>
</tbody>
</table>

The device_identify function adds a new device (instance) to an I/O bus. This function is used only by buses that cannot directly identify their children. Recall that autoconfiguration begins by identifying the child devices on each I/O bus. Modern buses can directly identify the devices that are connected to them. Older buses, such as ISA, have to use the device_identify routine provided by their associated drivers to identify their child devices (McKusick and Neville-Neil, 2005). You’ll learn how to associate a driver with an I/O bus shortly.

All identified child devices are passed to every Newbus driver’s device_probe function. A device_probe function tells the kernel whether its driver can handle the identified device.

Note that there may be more than one driver that can handle an identified child device. Thus, device_probe’s return value is used to specify how well its driver matches the identified device. The device_probe function that returns
the highest value denotes the best Newbus driver for the identified device. The following excerpt from `<sys/bus.h>` shows the constants used to indicate success (that is, a match):

```c
#define BUS_PROBE_SPECIFIC         0       /* Only I can use this device. */
#define BUS_PROBE_VENDOR           (-10)   /* Vendor-supplied driver. */
#define BUS_PROBE_DEFAULT          (-20)   /* Base OS default driver. */
#define BUS_PROBE_LOW_PRIORITY     (-40)   /* Older, less desirable driver. */
#define BUS_PROBE_GENERIC          (-100)  /* Generic driver for device. */
#define BUS_PROBE_HOOVER           (-500)  /* Driver for all devices on bus. */
#define BUS_PROBE_NOWILDCARD       (-2000000000) /* No wildcard matches. */
```

As you can see, success codes are values less than or equal to zero. The standard UNIX error codes (that is, positive values) are used as failure codes.

Once the best driver has been found to handle a device, its `device_attach` function is called. A `device_attach` function initializes a device and any essential software (for example, device nodes).

The `device_detach` function disconnects a driver from a device. This function should set the device to a sane state and release any resources that were allocated during `device_attach`.

A Newbus driver’s `device_shutdown`, `device_suspend`, and `device_resume` functions are called when the system is shut down, when its device is suspended, or when its device returns from suspension, respectively. These functions let a driver manage its device as these events occur.

### Device Method Table

A device method table, `device_method_t`, specifies which `device_foo` functions a Newbus driver implements. It is defined in the `<sys/bus.h>` header.

Here is an example device method table for a fictitious PCI device:

```c
static device_method_t foo_pci_methods[] = {
    /* Device interface. */
    {DEVMETHOD(device_probe, foo_pci_probe),
    {DEVMETHOD(device_attach, foo_pci_attach),
    {DEVMETHOD(device_detach, foo_pci_detach),
    {0, 0 }

    
};
```

As you can see, not every `device_foo` function has to be defined. If a `device_foo` function is undefined, the corresponding operation is unsupported.

Unsurprisingly, the `device_probe` and `device_attach` functions must be defined for every Newbus driver. For drivers on older buses, the `device_identify` function must also be defined.
**DRIVER_MODULE Macro**

The DRIVER_MODULE macro registers a Newbus driver with the system. This macro is defined in the `<sys/bus.h>` header. Here is its function prototype:

```c
#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/bus.h>
#include <sys/module.h>

DRIVER_MODULE(name, busname, driver_t driver, devclass_t devclass,
               modeventhand_t evh, void *arg);
```

The arguments expected by this macro are as follows.

**name**

The `name` argument is used to identify the driver.

**busname**

The `busname` argument specifies the driver’s I/O bus (for example, isa, pci, usb, and so on).

**driver**

The `driver` argument expects a filled-out `driver_t` structure. This argument is best understood with an example:

```c
static driver_t foo_pci_driver = {
    "foo_pci",
    foo_pci_methods,
    sizeof(struct foo_pci_softc)
};
```

Here, ① "foo_pci" is this example driver’s official name, ② `foo_pci_methods` is its device method table, and ③ `sizeof(struct foo_pci_softc)` is the size of its software context.

**devclass**

The `devclass` argument expects an uninitialized `devclass_t` variable, which will be used by the kernel for internal bookkeeping.

**evh**

The `evh` argument denotes an optional module event handler. Generally, we’ll always set `evh` to 0, because DRIVER_MODULE supplies its own module event handler.

**arg**

The `arg` argument is the `void *` argument for the module event handler specified by `evh`. If `evh` is set to 0, `arg` must be too.
Tying Everything Together

You now know enough to write your first Newbus driver. Listing 7-1 is a simple Newbus driver (based on code written by Murray Stokely) for a fictitious PCI device.

NOTE Take a quick look at this code and try to discern some of its structure. If you don’t understand all of it, don’t worry; an explanation follows.

```c
#include <sys/param.h>
#include <sys/module.h>
#include <sys/kernel.h>
#include <sys/systm.h>
#include <sys/conf.h>
#include <sys/uio.h>
#include <sys/bus.h>
#include <dev/pci/pcireg.h>
#include <dev/pci/pcivar.h>

struct foo_pci_softc {
    device_t        device;
    struct cdev     *cdev;
};

static d_open_t        foo_pci_open;
static d_close_t        foo_pci_close;
static d_read_t         foo_pci_read;
static d_write_t        foo_pci_write;

static struct cdevsw foo_pci_cdevsw = {
    .d_version =    D_VERSION,
    .d_open =       foo_pci_open,
    .d_close =      foo_pci_close,
    .d_read =       foo_pci_read,
    .d_write =      foo_pci_write,
    .d_name =       "foo_pci"
};

static devclass_t foo_pci_devclass;

static int
foo_pci_open(struct cdev *dev, int oflags, int devtype, struct thread *td) {
    struct foo_pci_softc *sc;
    sc = dev->si_drv1;
    device_printf(sc->device, "opened successfully\n");
    return (0);
}
```
static int
foo_pci_close(struct cdev *dev, int fflag, int devtype, struct thread *td)
{
    struct foo_pci_softc *sc;

    sc = dev->si_drv1;
    device_printf(sc->device, "closed\n");
    return (0);
}

static int
foo_pci_read(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct foo_pci_softc *sc;

    sc = dev->si_drv1;
    device_printf(sc->device, "read request = %dB\n", uio->uio_resid);
    return (0);
}

static int
foo_pci_write(struct cdev *dev, struct uio *uio, int ioflag)
{
    struct foo_pci_softc *sc;

    sc = dev->si_drv1;
    device_printf(sc->device, "write request = %dB\n", uio->uio_resid);
    return (0);
}

static struct _pcsid {
    uint32_t type;
    const char *desc;
} pci_ids[] = {
    { 0x1234abcd, "RED PCI Widget" },
    { 0x4321fedc, "BLU PCI Widget" },
    { 0x00000000, NULL }
};

static int
foo_pci_probe(device_t dev)
{
    uint32_t type = pci_get_devid(dev);
    struct _pcsid *ep = pci_ids;

    while (ep->type && ep->type != type)
        ep++;
    if (ep->desc) {
        device_set_desc(dev, ep->desc);
        return (BUS_PROBE_DEFAULT);
    }

    return (ENXIO);
}
static int
foo_pci_attach(device_t dev)
{
    struct foo_pci_softc *sc = device_get_softc(dev);
    int unit = device_get_unit(dev);

    sc->device = dev;
    sc->cdev = make_dev(&foo_pci_cdevsw, unit, UID_ROOT, GID_WHEEL,
                        0600, "foo_pci%d", unit);
    sc->cdev->si_drv1 = sc;

    return (0);
}

static int
foo_pci_detach(device_t dev)
{
    struct foo_pci_softc *sc = device_get_softc(dev);

    destroy_dev(sc->cdev);
    return (0);
}

static device_method_t foo_pci_methods[] = {
    /* Device interface. */
    DEVMETHOD(device_probe,         foo_pci_probe),
    DEVMETHOD(device_attach,        foo_pci_attach),
    DEVMETHOD(device_detach,        foo_pci_detach),
    { 0, 0 }
};

static driver_t foo_pci_driver = {
    "foo_pci",
    foo_pci_methods,
    sizeof(struct foo_pci_softc)
};

DRIVER_MODULE(foo_pci, pci, foo_pci_driver, foo_pci_devclass, 0, 0);

Listing 7-1: foo_pci.c

This driver begins by defining its software context, which will maintain a pointer to its device and the cdev returned by the make_dev call.

Next, its character device switch table is defined. This table contains four d_foo functions named foo_pci_open, foo_pci_close, foo_pci_read, and foo_pci_write. I'll describe these functions in “d_foo Functions” on page 121.

Then a devclass_t variable is declared. This variable is passed to the DRIVER_MODULE macro as its devclass argument.

Finally, the d_foo and device_foo functions are defined. These functions are described in the order they would execute.
**foo_pci_probe Function**

The `foo_pci_probe` function is the `device_probe` implementation for this driver. Before I walk through this function, a description of the `pci_ids` array (found around the middle of Listing 7-1) is needed.

```c
static struct _pcsid {
    uint32_t  type;
    const char *desc;
} pci_ids[] = {
    { 0x1234abcd, "RED PCI Widget" },
    { 0x4321fedc, "BLU PCI Widget" },
    { 0x00000000, NULL }
};
```

This array is composed of three `_pcsid` structures. Each `_pcsid` structure contains a PCI ID and a description of the PCI device. As you might have guessed, `pci_ids` lists the devices that Listing 7-1 supports.

Now that I've described `pci_ids`, let’s walk through `foo_pci_probe`.

```c
static int foo_pci_probe(device_t dev)
{
    uint32_t type = pci_get_devid(dev);
    struct _pcsid *ep = pci_ids;

    while (ep->type && ep->type != type)
        ep++;
    if (ep->desc) {
        device_set_desc(dev, ep->desc);
        return (BUS_PROBE_DEFAULT);
    }
    return (ENXIO);
}
```

Here, `dev` describes an identified device found on the PCI bus. So this function begins by obtaining the PCI ID of `dev`. Then it determines if `dev`’s PCI ID is listed in `pci_ids`. If it is, `dev`’s verbose description is set and the success code `BUS_PROBE_DEFAULT` is returned.

**NOTE** The verbose description is printed to the system console when `foo_pci_attach` executes.

**foo_pci_attach Function**

The `foo_pci_attach` function is the `device_attach` implementation for this driver. Here is its function definition (again):

```c
static int foo_pci_attach(device_t dev)
{
    struct foo_pci_softc *sc = device_get_softc(dev);
```
int unit = device_get_unit(dev);

sc->device = dev;
sc->cdev = make_dev(&foo_pci_cdevsw, unit, UID_ROOT, GID_WHEEL,
       0600, "foo_pci%d", unit);
sc->cdev->si_drv1 = sc;

return (0);
}

Here, dev describes a device under this driver’s control. Thus, this
function starts by getting dev’s software context and unit number.
Then a character device node is created and the variables sc->device and
sc->cdev->si_drv1 are set to dev and sc, respectively.

NOTE The d_foo functions (described next) use sc->device and cdev->si_drv1 to gain access
to dev and sc.

d_foo Functions

Because every d_foo function in Listing 7-1 just prints a debug message (that
is to say, they’re all basically the same), I’m only going to walk through one
of them: foo_pci_open.

static int
foo_pci_open(struct cdev *dev, int oflags, int devtype, struct thread *td)
{
    struct foo_pci_softc *sc;

    sc = dev->si_drv1;
    device_printf(sc->device, "opened successfully\n");
    return (0);
}

Here, dev is the cdev returned by the make_dev call in foo_pci_attach. So,
this function first obtains its software context. Then it prints a debug
message.

foo_pci_detach Function

The foo_pci_detach function is the device_detach implementation for this
driver. Here is its function definition (again):

static int
foo_pci_detach(device_t dev)
{
    struct foo_pci_softc *sc = device_get_softc(dev);

    destroy_dev(sc->cdev);
    return (0);
}
Here, \( \text{dev} \) describes a device under this driver's control. Thus, this function simply obtains \( \text{dev}'s \) software context to destroy its device node.

**Don’t Panic**

Now that we’ve discussed Listing 7-1, let’s give it a try:

```bash
$ sudo kldload ./foo_pci.ko
$ kldstat
Id Refs Address    Size     Name
 1 3 0xc0400000 c9f490   kernel
 2 1 0xc3af0000 2000     foo_pci.ko
$ ls -l /dev/foo*
ls: /dev/foo*: No such file or directory
```

Of course, it fails miserably, because `foo_pci_probe` is probing for fictitious PCI devices. Before concluding this chapter, one additional topic bears mentioning.

**Hardware Resource Management**

As part of configuring and operating devices, a driver might need to manage hardware resources, such as interrupt-request lines (IRQs), I/O ports, or I/O memory (McKusick and Neville-Neil, 2005). Naturally, Newbus includes several functions for doing just that.

```c
#include <sys/param.h>
#include <sys/bus.h>

#include <machine/bus.h>
#include <sys/rman.h>
#include <machine/resource.h>

struct resource *
bus_alloc_resource(device_t dev, int type, int *rid, u_long start,
       u_long end, u_long count, u_int flags);

struct resource *
bus_alloc_resource_any(device_t dev, int type, int *rid,
       u_int flags);

int
bus_activate_resource(device_t dev, int type, int rid,
       struct resource *r);

int
bus_deactivate_resource(device_t dev, int type, int rid,
       struct resource *r);

int
bus_release_resource(device_t dev, int type, int rid,
       struct resource *r);
```
The `bus_alloc_resource` function allocates hardware resources for a specific device to use. If successful, a `struct resource` pointer is returned; otherwise, `NULL` is returned. This function is normally called during `device_attach`. If it is called during `device_probe`, all allocated resources must be released (via `bus_release_resource`) before returning. Most of the arguments for `bus_alloc_resource` are common to the other hardware resource management functions. These arguments are described in the next few paragraphs.

The `dev` argument is the device that requires ownership of the hardware resource(s). Before allocation, resources are owned by the parent bus.

The `type` argument represents the type of resource `dev` wants allocated. Valid values for this argument are listed in Table 7-2.

The `rid` argument expects a resource ID (RID). If `bus_alloc_resource` is successful, a RID is returned in `rid` that may differ from what you passed. You'll learn more about RIDs later.

The `start` and `end` arguments are the start and end addresses of the hardware resource(s). To employ the default bus values, simply pass `0ul` as `start` and `~0ul` as `end`.

The `count` argument denotes the size of the hardware resource(s). If you used the default bus values for `start` and `end`, `count` is used only if it is larger than the default bus value.

The `flags` argument details the characteristics of the hardware resource. Valid values for this argument are listed in Table 7-3.

The `bus_alloc_resource_any` function is a convenience wrapper for `bus_alloc_resource` that sets `start`, `end`, and `count` to their default bus values.

The `bus_activate_resource` function activates a previously allocated hardware resource. Naturally, resources must be activated before they can be used. Most drivers simply pass `RF_ACTIVE` to `bus_alloc_resource` or `bus_alloc_resource_any` to avoid calling `bus_activate_resource`.

### Table 7-2: Symbolic Constants for Hardware Resources

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS_RES_IRQ</td>
<td>Interrupt-request line</td>
</tr>
<tr>
<td>SYS_RES_IOPORT</td>
<td>I/O port</td>
</tr>
<tr>
<td>SYS_RES_MEMORY</td>
<td>I/O memory</td>
</tr>
</tbody>
</table>

### Table 7-3: `bus_alloc_resource` Symbolic Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RF_ALLOCATED</td>
<td>Allocate hardware resource, but don’t activate it</td>
</tr>
<tr>
<td>RF_ACTIVE</td>
<td>Allocate hardware resource and activate resource automatically</td>
</tr>
<tr>
<td>RF_SHAREABLE</td>
<td>Hardware resource permits contemporaneous sharing; you should always set this flag, unless the resource cannot be shared</td>
</tr>
<tr>
<td>RF_TIMESHARE</td>
<td>Hardware resource permits time-division sharing</td>
</tr>
</tbody>
</table>
The `bus_deactivate_resource` function deactivates a hardware resource. This function is primarily used in bus drivers (so we’ll never call it).

The `bus_release_resource` function releases a previously allocated hardware resource. Of course, the resource cannot be in use on release. If successful, 0 is returned; otherwise, the kernel panics.

**NOTE** We’ll cover an example that employs IRQs in Chapters 8 and 9, and I’ll go over an example that requires I/O ports and I/O memory in Chapters 10 and 11.

**Conclusion**

This chapter introduced you to the basics of Newbus driver development—working with real hardware. The remainder of this book builds upon the concepts described here to complete your understanding of Newbus.