

FreeBSD Device Driver Writer's Guide

Eric L. Hernes, erich@rrnet.com

Wednesday, May 29, 1996

Abstract

This document describes how to add a device driver to FreeBSD. It is *not* intended to be a tutorial on UNIX device drivers in general. It is intended for device driver authors, familiar with the UNIX device driver model, to work on FreeBSD.

1. Overview

The FreeBSD kernel is very well documented, unfortunately it's all in 'C'.

2. Types of drivers.

2.1 Character

2.1.1 Data Structures

```
struct cdevsw Structure
```

2.1.2 Entry Points

2.1.2.1 *d_open()*

d_open() takes several arguments, the formal list looks something like:

```
int
d_open(dev_t dev, int flag, int mode, struct proc *p)
```

d_open() is called on *every* open of the device.

The *dev* argument contains the major and minor number of the device opened. These are available through the macros `major()` and `minor()`

The *flag* and *mode* arguments are as described in the `open(2)` manual page. It is recommended that you check these for access modes in `<sys/fcntl.h>` and do what is required. For example if *flag* is `(O_NONBLOCK | O_EXLOCK)` the open should fail if either it would block, or exclusive access cannot be granted.

The *p* argument contains all the information about the current process.

2.1.2.2 `d_close()`

`d_close()` takes the same argument list as `d_open()`:

```
int
d_close(dev_t dev , int flag , int mode , struct proc *p)
```

`d_close()` is only called on the last close of your device (per minor device). For example in the following code fragment, `d_open()` is called 3 times, but `d_close()` is called only once.

```
...
    fd1=open("/dev/mydev", O_RDONLY);
    fd2=open("/dev/mydev", O_RDONLY);
    fd3=open("/dev/mydev", O_RDONLY);
...
    <useful stuff with fd1, fd2, fd3 here>
...
    close(fd1);
    close(fd2);
    close(fd3);
...

```

The arguments are similar to those described above for `d_open()`.

2.1.2.3 `d_read()` and `d_write()`

`d_read()` and `d_write` take the following argument lists:

```
int
d_read(dev_t dev, struct uio *uio, int flat)
int
d_write(dev_t dev, struct uio *uio, int flat)
```

The `d_read()` and `d_write()` entry points are called when `read(2)` and `write(2)` are called on your device from user-space. The transfer of data can be handled through the kernel support routine `uiomove()`.

2.1.2.4 `d_ioctl()`

It's argument list is as follows:

```
int
d_ioctl(dev_t dev, int cmd, caddr_t arg, int flag, struct proc *p)
```

`d_ioctl()` is a catch-all for operations which don't make sense in a read/write paradigm. Probably the most famous of all `ioctl`'s is on tty devices, through `stty(1)`. The `ioctl` entry point is called from `ioctl()` in `sys/kern/sys_generic.c`

There are four different types of `ioctl`'s which can be implemented. `<sys/ioccom.h>` contains convenience macros for defining these `ioctls`.

`_IO(g,n)` for control type operations.

`_IOR(g,n,t)` for operations that read data from a device.

`_IOW(g,n,t)` for operations that write data to a device.

`_IOWR(g,n,t)` for operations that write to a device, and then read data back.

Here `g` refers to a *group*. This is an 8-bit value, typically indicative of the device; for example, 't' is used in tty `ioctls`. `n` refers to the number of the `ioctl` within the group. On SCO, this number alone denotes the `ioctl`. `t` is the data type which will get passed to the driver; this gets handed to a `sizeof()` operator in the kernel. The `ioctl()` system call will either `copyin()` or `copyout()` or both for your driver, then hand you a pointer to the data structure in the `arg` argument of the `d_ioctl` call. Currently the data size is limited to one page (4k on the i386).

2.1.2.5 d_stop()**2.1.2.6 d_reset()****2.1.2.7 d_devtotty()****2.1.2.8 d_poll() (3.0+) or d_select() (2.2)**

d_poll()'s argument list is as follows:

```
void
d_poll(dev_t dev, int events, struct proc *p)
```

d_poll() is used to find out if a device is ready for IO. An example is waiting for data to become available from the network, or waiting for the user to press a key. This correspond to the poll() call in userland.

The d_poll() call should check for the events specified in the event mask. If none of the requested events are active, but they may become active later, it should record this for the kernel's later perusal. d_poll() does this by calling selrecord() with a selinfo structure for this device. The sum of all these activities look something like this:

```
static struct my_softc {
    struct queue rx_queue; /* As example only - not required */
    struct queue tx_queue; /* As example only - not required */
    struct selinfo selp; /* Required */
} my_softc[NMYDEV];

...

static int
mydevpoll(dev_t dev, int events, struct proc *p)
{
    int revents = 0; /* Events we found */
    int s;
    struct my_softc *sc = &my_softc[dev];

    /* We can only check for IN and OUT */
    if ((events & (POLLIN|POLLOUT)) == 0)
        return(POLLNVAL);

    s = splhigh();
    /* Writes are if the transmit queue can take them */
    if ((events & POLLOUT) &&
        !IF_QFULL(sc->tx_queue))
        revents |= POLLOUT;
    /* ... while reads are OK if we have any data */
    if ((events & POLLIN) &&
        !IF_QEMPTY(sc->rx_queue))
        revents |= POLLIN;
    if (revents == 0)
        selrecord(p, &sc->selp);
    splx(s);
    return revents;
}
```

d_select() is used in 2.2 and previous versions of FreeBSD. Instead of 'events' it take a single int 'rw', which can be FREAD for reads (as in POLLIN above), FWRITE for write (as in POLLOUT above), and 0 for 'exception' - something exceptional happened, like a card being inserted or removed for the pccard driver.

For select, the above code fragment would look like this:

```

static int
mydevselect(dev_t dev, int rw, struct proc *p)
{
    int ret = 0;
    int s;
    struct my_softc *sc = &my_softc[dev];

    s = splhigh();
    switch (rw) {
    case FWRITE:
        /* Writes are OK if the transmit queue can take them */
        if (!IF_QFULL(sc->tx_queue))
            ret = 1;
        break;
    case FREAD:
        /* ... while reads are OK if we have any data */
        if (!IF_QEMPTY(sc->rx_queue))
            ret = 1;
        break;
    case 0:
        /* This driver never get any exceptions */
        break;
    }
    if (ret == 0)
        selrecord(p, &sc->sel);
    splx(s);
    return(revents);
}

```

2.1.2.9 *d_mmap()*

2.1.2.10 *d_strategy()*

d_strategy()'s argument list is as follows:

```

void
d_strategy(struct buf *bp)

```

d_strategy() is used for devices which use some form of scatter-gather io. It is most common in a block device. This is significantly different than the System V model, where only the block driver performs scatter-gather io. Under BSD, character devices are sometimes requested to perform scatter-gather io via the *readv()* and *writv()* system calls.

2.1.3 Header Files

2.2 Block

2.2.1 Data Structures

```

struct bdevsw Structure
struct buf Structure

```

2.2.2 Entry Points

2.2.2.1 *d_open()*

Described in the Character device section.

2.2.2.2 *d_close()*

Described in the Character device section.

2.2.2.3 *d_strategy()*

Described in the Character device section.

2.2.2.4 *d_ioctl()*

Described in the Character device section.

2.2.2.5 *d_dump()*

2.2.2.6 *d_psize()*

2.2.3 Header Files

2.3 Network

2.3.1 Data Structures

`struct ifnet` Structure

2.3.2 Entry Points

2.3.2.1 *if_init()*

2.3.2.2 *if_output()*

2.3.2.3 *if_start()*

2.3.2.4 *if_done()*

2.3.2.5 *if_ioctl()*

2.3.2.6 *if_watchdog()*

2.3.3 Header Files

2.4 Line Discipline

2.4.1 Data Structures

`struct linesw` Structure

2.4.2 Entry Points

2.4.2.1 *l_open()*

2.4.2.2 *l_close()*

2.4.2.3 *l_read()*

2.4.2.4 *l_write()*

2.4.2.5 *l_ioctl()*

2.4.2.6 *l_rint()*

2.4.2.7 *l_start()*

2.4.2.8 *l_modem()*

2.4.3 Header Files

3. Supported Busses

3.1 ISA -- Industry Standard Architecture

3.1.1 Data Structures

3.1.1.1 *struct isa_device Structure*

This structure is required, but generally it is created by `config(8)` from the kernel configuration file. It is required on a per-device basis, meaning that if you have a driver which controls two serial boards, you will have two `isa_device` structures. If you build a device as an LKM, you must create your own `isa_device` structure to reflect your configuration. (lines 85 - 131 in `pcaudio_lkm.c`) There is nearly a direct mapping between the config file and the `isa_device` structure. The definition from `/usr/src/sys/i386/isa/isa_device.h` is:

```
struct isa_device {
    int     id_id;           /* device id */
    struct  isa_driver *id_driver;
    int     id_iobase;      /* base i/o address */
    u_short id_irq;        /* interrupt request */
    short   id_drq;         /* DMA request */
    caddr_t id_maddr;       /* physical i/o memory address on bus (if any)*/
    int     id_msize;       /* size of i/o memory */
    inthand2_t *id_intr;    /* interrupt interface routine */
    int     id_unit;        /* unit number */
    int     id_flags;       /* flags */
    int     id_scsiid;      /* scsi id if needed */
    int     id_alive;       /* device is present */
#define RI_FAST 1          /* fast interrupt handler */
    u_int   id_ri_flags;    /* flags for register_intr() */
    int     id_reconfig;    /* hot eject device support (such as PCMCIA) */
    int     id_enabled;     /* is device enabled */
    int     id_conflicts;   /* we're allowed to conflict with things */
    struct  isa_device *id_next; /* used in isa_devlist in userconfig() */
};
```

3.1.1.2 *struct isa_driver Structure*

This structure is defined in `"/usr/src/sys/i386/isa/isa_device.h"`. These are required on a per-driver basis. The definition is:

```
struct isa_driver {
    int     (*probe) __P((struct isa_device *idp));
                                     /* test whether device is present */
    int     (*attach) __P((struct isa_device *idp));
                                     /* setup driver for a device */
    char    *name;                    /* device name */
    int     sensitive_hw;              /* true if other probes confuse us */
};
```

This is the structure used by the probe/attach code to detect and initialize your device. The `probe` member is a pointer to your device probe function; the `attach` member is a pointer to your attach function. The `name` member is a character pointer to the two or three letter name for your driver. This is the name reported during the probe/attach process (and probably also in `lsdev(8)`). The `sensitive_hw` member is a flag which helps the probe code determine probing order.

A typical instantiation is:

```
struct isa_driver    mcd_driver = { mcd_probe, mcd_attach, "mcd" };
```

3.1.2 Entry Points

3.1.2.1 *probe()*

`probe()` takes a `struct isa_device` pointer as an argument and returns an int. The return value is “zero” or “non-zero” as to the absence or presence of your device. This entry point may (and probably should) be declared as `static` because it is accessed via the `probe` member of the `struct isa_driver` structure. This function is intended to detect the presence of your device only; it should not do any configuration of the device itself.

3.1.2.2 *attach()*

attach() also takes a `struct isa_device` pointer as an argument and returns an `int`. The return value is also “zero” or “non-zero” indicating whether or not the attach was successful. This function is intended to do any special initialization of the device as well as confirm that the device is usable. It too should be declared `static` because it is accessed through the `attach` member of the `isa_driver` structure.

3.1.3 Header Files

3.2 EISA -- Extended Industry Standard Architecture

3.2.1 Data Structures

```
struct eisa_dev Structure
struct isa_driver Structure
```

3.2.2 Entry Points

3.2.2.1 *probe()*

Described in the ISA device section.

3.2.2.2 *attach()*

Described in the ISA device section.

3.2.3 Header Files

3.3 PCI -- Peripheral Computer Interconnect

3.3.1 Data Structures

```
struct pci_device Structure
```

`name`: The short device name.

`probe`: Checks if the driver can support a device with this type. The tag may be used to get more info with `pci_read_conf()`. See below. It returns a string with the device's name, or a NULL pointer, if the driver cannot support this device.

`attach`: Allocate a control structure and prepare it. This function may use the PCI mapping functions. See below. (`configuration id`) or `type`.

`count`: A pointer to a unit counter. It's used by the PCI configurator to allocate unit numbers.

3.3.2 Entry Points

3.3.2.1 *probe()*

3.3.2.2 *attach()*

3.3.2.3 *shutdown()*

3.3.3 Header Files

3.4 SCSI -- Small Computer Systems Interface

3.4.1 Data Structures

```
struct scsi_adapter Structure
struct scsi_device Structure
struct scsi_ctlr_config Structure
```

```
struct scsi_device_config Structure
struct scsi_link Structure
```

3.4.2 Entry Points

3.4.2.1 *attach()*

3.4.2.2 *init()*

3.4.3 Header Files

3.5 PCCARD (PCMCIA)

3.5.1 Data Structures

```
struct slot_cont Structure
struct pccard_drv Structure
struct pccard_dev Structure
struct slot Structure
```

3.5.2 Entry Points

3.5.2.1 *handler()*

3.5.2.2 *unload()*

3.5.2.3 *suspend()*

3.5.2.4 *init()*

3.5.3 Header Files

a. <pccard/slot.h>"

4. Linking Into the Kernel.

In FreeBSD, support for the ISA and EISA busses is i386 specific. While FreeBSD itself is presently available on the i386 platform, some effort has been made to make the PCI, PCCARD, and SCSI code portable. The ISA and EISA specific code resides in /usr/src/sys/i386/isa and /usr/src/sys/i386/eisa respectively. The machine independent PCI, PCCARD, and SCSI code reside in /usr/src/sys/{pci,pccard,scsi}. The i386 specific code for these reside in /usr/src/sys/i386/{pci,pccard,scsi}.

In FreeBSD, a device driver can be either binary or source. There is no "official" place for binary drivers to reside. BSD/OS uses something like sys/i386/OBJ. Since most drivers are distributed in source, the following discussion refers to a source driver. Binary only drivers are sometimes provided by hardware vendors who wish to maintain the source as proprietary.

A typical driver has the source code in one c-file, say dev.c. The driver also can have some include files; devreg.h typically contains public device register declarations, macros, and other driver specific declarations. Some drivers call this devvar.h instead. Some drivers, such as the dgb (for the Digiboard PC/Xe), require microcode to be loaded onto the board. For the dgb driver the microcode is compiled and dumped into a header file ala file2c(1).

If the driver has data structures and ioctl's which are specific to the driver/device, and need to be accessible from user-space, they should be put in a separate include file which will reside in /usr/include/machine/ (some of these reside in /usr/include/sys/). These are typically named something like ioctl_dev.h or devio.h.

If a driver is being written which, from user space is identical to a device which already exists,

care should be taken to use the same ioctl interface and data structures. For example, from user space, a SCSI CDROM drive should be identical to an IDE cdrom drive; or a serial line on an intelligent multiport card (Digiboard, Cyclades, ...) should be identical to the sio devices. These devices have a fairly well defined interface which should be used.

There are two methods for linking a driver into the kernel, static and the LKM model. The first method is fairly standard across the *BSD family. The other method was originally developed by Sun (I believe), and has been implemented into BSD using the Sun model. I don't believe that the current implementation uses any Sun code.

4.1 Standard Model

The steps required to add your driver to the standard FreeBSD kernel are

- Add to the driver list
- Add an entry to the [bc]devsw
- Add the driver entry to the kernel config file
- config(8), compile, and install the kernel
- make required nodes.
- reboot.

4.1.1 Adding to the driver list.

The standard model for adding a device driver to the Berkeley kernel is to add your driver to the list of known devices. This list is dependent on the CPU architecture. If the device is not i386 specific (PCCARD, PCI, SCSI), the file is in `"/usr/src/sys/conf/files"`. If the device is i386 specific, use `"/usr/src/sys/i386/conf/files.i386"`. A typical line looks like:

```
i386/isa/joy.c          optional      joy      device-driver
```

The first field is the pathname of the driver module relative to `/usr/src/sys`. For the case of a binary driver the path would be something like `"i386/OBJ/joy.o"`.

The second field tells config(8) that this is an optional driver. Some devices are required for the kernel to even be built.

The third field is the name of the device.

The fourth field tells config that it's a device driver (as opposed to just optional). This causes config to create entries for the device in some structures in `/usr/src/sys/compile/KERNEL/ioconf.c`.

It is also possible to create a file `"/usr/src/sys/i386/conf/files.KERNEL"` whose contents will override the default files.i386, but only for the kernel "KERNEL".

4.1.2 Make room in conf.c

Now you must edit `"/usr/src/sys/i386/i386/conf.c"` to make an entry for your driver. Somewhere near the top, you need to declare your entry points. The entry for the joystick driver is:

```

#include "joy.h"
#if NJOY > 0
d_open_t      joyopen;
d_close_t     joyclose;
d_rdwr_t     joyread;
d_ioctl_t     joyioctl;
#else
#define joyopen      nxopen
#define joyclose     nxclose
#define joyread      nxread
#define joyioctl     nxioctl
#endif

```

This either defines your entry points, or null entry points which will return ENXIO when called (the #else clause).

The include file “joy.h” is automatically generated by config(8) when the kernel build tree is created. This usually has only one line like:

```
#define NJOY 1
```

or

```
#define NJOY 0
```

which defines the number of your devices in your kernel.

You must additionally add a slot to either cdevsw[], or to bdevsw[], depending on whether it is a character device or a block device, or both if it is a block device with a raw interface. The entry for the joystick driver is:

```

/* open, close, read, write, ioctl, stop, reset, ttys, select, mmap, strat */
struct cdevsw  cdevsw[] =
{
    ...
    { joyopen,      joyclose,      joyread,      nowrite,      /*51*/
      joyioctl,    nostop,        nullreset,    nodevtotty, /*joystick */
      seltrue,     nommap,        NULL},
    ...
}

```

Order is what determines the major number of your device. Which is why there will always be an entry for your driver, either null entry points, or actual entry points. It is probably worth noting that this is significantly different from SCO and other system V derivatives, where any device can (in theory) have any major number. This is largely a convenience on FreeBSD, due to the way device nodes are created. More on this later.

4.1.3 Adding your device to the config file.

This is simply adding a line describing your device. The joystick description line is:

```
device      joy0      at isa? port "IO_GAME"
```

This says we have a device called “joy0” on the isa bus using io-port “IO_GAME” (IO_GAME is a macro defined in /usr/src/sys/i386/isa/isa.h).

A slightly more complicated entry is for the “ix” driver:

```
device ix0 at isa? port 0x300 net irq 10 iomem 0xd0000 iosiz 32768 vector ixintr
```

This says that we have a device called ‘ix0’ on the ISA bus. It uses io-port 0x300. It’s interrupt will be masked with other devices in the network class. It uses interrupt 10. It uses 32k of shared

memory at physical address 0xd0000. It also defines it's interrupt handler to be "ixintr()"

4.1.4

the kernel."

Now with our config file in hand, we can create a kernel compile directory. This is done by simply typing:

```
# config KERNEL
```

where KERNEL is the name of your config file. Config creates a compile tree for you kernel in /usr/src/sys/compile/KERNEL. It creates the Makefile, some .c files, and some .h files with macros defining the number of each device in your kernel.

Now you can go to the compile directory and build. Each time you run config, your previous build tree will be removed, unless you config with a -n. If you have config'ed and compiled a GENERIC kernel, you can "make links" to avoid compiling a few files on each iteration. I typically run

```
# make depend links all
```

followed by a "make install" when the kernel is done to my liking.

4.1.5 Making device nodes.

On FreeBSD, you are responsible for making your own device nodes. The major number of your device is determined by the slot number in the device switch. Minor number is driver dependent, of course. You can either run the mknod's from the command line, or add a section to /dev/MAKEDEV.local, or even /dev/MAKEDEV to do the work. I sometimes create a MAKEDEV.dev script that can be run stand-alone or pasted into /dev/MAKEDEV.local

4.1.6 Reboot.

This is the easy part. There are a number of ways to do this, reboot, fastboot, shutdown -r, cycle the power, etc. Upon bootup you should see your XXprobe() called, and if all is successful, your XXattach() too.

4.2 Loadable Kernel Module (LKM)

There are really no defined procedures for writing an LKM driver. The following is my own conception after experimenting with the LKM device interface and looking at the standard device driver model, this is one way of adding an LKM interface to an existing driver without touching the original driver source (or binary). It is recommended though, that if you plan to release source to your driver, the LKM specific parts should be part of the driver itself, conditionally compiled on the LKM macro (i.e. #ifdef LKM).

This section will focus on writing the LKM specific part of the driver. We will assume that we have written a driver which will drop into the standard device driver model, which we would now like to implement as an LKM. We will use the pcaudio driver as a sample driver, and develop an LKM front-end. The source and makefile for the pcaudio LKM, "pcaudio_lkm.c" and "Makefile", should be placed in /usr/src/lkm/pcaudio. What follows is a breakdown of pcaudio_lkm.c.

Lines 17 - 26

```
-- This includes the file "pca.h" and conditionally compiles the rest of the LKM on whether or not we have a pcaudio device defined. This mimics the behavior of config. In a standard device driver, config(8) generates the pca.h file from the number pca devices in the config file.
```

```

17  /*
18  * figure out how many devices we have..
19  */
20
21  #include "pca.h"
22
23  /*
24  * if we have at least one ...
25  */
26  #if NPCA > 0

```

Lines 27 - 37

-- Includes required files from various include directories.

```

27  #include <sys/param.h>
28  #include <sys/system.h>
29  #include <sys/exec.h>
30  #include <sys/conf.h>
31  #include <sys/sysent.h>
32  #include <sys/lkm.h>
33  #include <sys/errno.h>
34  #include <i386/isa/isa_device.h>
35  #include <i386/isa/isa.h>
36
37

```

Lines 38 - 51

-- Declares the device driver entry points as external.

```

38  /*
39  * declare your entry points as externs
40  */
41
42  extern int pcaprobe(struct isa_device *);
43  extern int pcaattach(struct isa_device *);
44  extern int pcaopen(dev_t, int, int, struct proc *);
45  extern int pcaclose(dev_t, int, int, struct proc *);
46  extern int pcawrite(dev_t, struct uio *, int);
47  extern int pcaiioctl(dev_t, int, caddr_t);
48  extern int pcaselect(dev_t, int, struct proc *);
49  extern void pcaintr(struct clockframe *);
50  extern struct isa_driver pcdriver;
51

```

Lines 52 - 70

-- This is creates the device switch entry table for your driver. This table gets swapped wholesale into the system device switch at the location specified by your major number. In the standard model, these are in /usr/src/sys/i386/i386/conf.c. NOTE: you cannot pick a device major number higher than what exists in conf.c, for example at present, conf.c rev 1.85, there are 67 slots for character devices, you cannot use a (character) major device number 67 or greater, without first reserving space in conf.c.

```
52 /*
53  * build your device switch entry table
54  */
55
56 static struct cdevsw pcacdevsw = {
57     (d_open_t *)      pcaopen, /* open */
58     (d_close_t *)    pcaclose, /* close */
59     (d_rdwr_t *)     enodev, /* read */
60     (d_rdwr_t *)     pcawrite, /* write */
61     (d_ioctl_t *)    pcaioctl, /* ioctl */
62     (d_stop_t *)     enodev, /* stop?? */
63     (d_reset_t *)    enodev, /* reset */
64     (d_ttycv_t *)    enodev, /* ttys */
65     (d_select_t *)   pcaselect, /* select */
66     (d_mmap_t *)     enodev, /* mmap */
67     (d_strategy_t *) enodev /* strategy */
68 };
69
70
```

Lines 71 - 131

-- This section is analogous to the config file declaration of your device. The members of the `isa_device` structure are filled in by what is known about your device, I/O port, shared memory segment, etc. We will probably never have a need for two `pcaudio` devices in the kernel, but this example shows how multiple devices can be supported.

```

71 /*
72  * this lkm arbitrarily supports two
73  * instantiations of the pc-audio device.
74  *
75  * this is for illustration purposes
76  * only, it doesn't make much sense
77  * to have two of these beasts...
78  */
79
80
81 /*
82  * these have a direct correlation to the
83  * config file entries...
84  */
85 struct isa_device pcadev[NPCA] = {
86  {
87     11,          /* device id */
88     &pcadriver, /* driver pointer */
89     IO_TIMER1,  /* base io address */
90     -1,         /* interrupt */
91     -1,         /* dma channel */
92     (caddr_t)-1, /* physical io memory */
93     0,          /* size of io memory */
94     pcaintr,    /* interrupt interface */
95     0,          /* unit number */
96     0,          /* flags */
97     0,          /* scsi id */
98     0,          /* is alive */
99     0,          /* flags for register_intr */
100    0,          /* hot eject device support */
101    1           /* is device enabled */
102  },
103  #if NPCA >1
104  {
105
106  /*
107   * these are all zeros, because it doesn't make
108   * much sense to be here
109   * but it may make sense for your device
110   */
111
112     0,          /* device id */
113     &pcadriver, /* driver pointer */
114     0,          /* base io address */
115     -1,         /* interrupt */
116     -1,         /* dma channel */
117     -1,         /* physical io memory */
118     0,          /* size of io memory */
119     NULL,       /* interrupt interface */
120     1,          /* unit number */
121     0,          /* flags */
122     0,          /* scsi id */
123     0,          /* is alive */
124     0,          /* flags for register_intr */
125     0,          /* hot eject device support */
126     1           /* is device enabled */
127  },
128  #endif
129  };
130 };
131

```

Lines 132 - 139

-- This calls the C-preprocessor macro MOD_DEV, which sets up an LKM device driver, as opposed to an LKM filesystem, or an LKM system call.

```

132  /*
133  * this macro maps to a function which
134  * sets the LKM up for a driver
135  * as opposed to a filesystem, system call, or misc
136  * LKM.
137  */
138  MOD_DEV("pcaudio_mod", LM_DT_CHAR, 24, &pcacdevsw);
139

```

Lines 140 - 168

-- This is the function which will be called when the driver is loaded. This function tries to work like `sys/i386/isa/isa.c` which does the probe/attach calls for a driver at boot time. The biggest trick here is that it maps the physical address of the shared memory segment, which is specified in the `isa_device` structure to a kernel virtual address. Normally the physical address is put in the config file which builds the `isa_device` structures in `/usr/src/sys/compile/KERNEL/ioconf.c`. The probe/attach sequence of `/usr/src/sys/isa/isa.c` translates the physical address to a virtual one so that in your probe/attach routines you can do things like

```
(int *)id->id_maddr = something;
```

and just refer to the shared memory segment via pointers.

```

140  /*
141  * this function is called when the module is
142  * loaded; it tries to mimic the behavior
143  * of the standard probe/attach stuff from
144  * isa.c
145  */
146  int
147  pcaload(){
148      int i;
149      uprintf("PC Audio Driver Loaded\n");
150      for (i=0; i<NPCA; i++){
151          /*
152           * this maps the shared memory address
153           * from physical to virtual, to be
154           * consistent with the way
155           * /usr/src/sys/i386/isa.c handles it.
156           */
157          pcadev[i].id_maddr -=0xa0000;
158          pcadev[i].id_maddr += atdevbase;
159          if ((*pcadriver.probe)(pcadev+i)) {
160              (*(pcadriver.attach))(pcadev+i);
161          } else {
162              uprintf("PC Audio Probe Failed\n");
163              return(1);
164          }
165      }
166      return 0;
167  }
168

```

Lines 169 - 179

-- This is the function called when your driver is unloaded; it just displays a message to that effect.

```
169 /*
170  * this function is called
171  * when the module is unloaded
172  */
173
174 int
175 pcaunload(){
176     uprintf("PC Audio Driver Unloaded\n");
177     return 0;
178 }
179
```

Lines 180 - 190

-- This is the entry point which is specified on the command line of the modload. By convention it is named `<dev>_mod`. This is how it is defined in `bsd.lkm.mk`, the makefile which builds the LKM. If you name your module following this convention, you can do “make load” and “make unload” from `/usr/src/lkm/pcaudio`.

Note: this has gone through *many* revisions from release 2.0 to 2.1. It may or may not be possible to write a module which is portable across all three releases.

```
180 /*
181  * this is the entry point specified
182  * on the modload command line
183  */
184
185 int
186 pcaudio_mod(struct lkm_table *lkmtpl, int cmd, int ver)
187 {
188     DISPATCH(lkmtpl, cmd, ver, pcaload, pcaunload, nosys);
189 }
190
191 #endif /* NICP > 0 */
```

4.3 Device Type Idiosyncrasies

4.3.1 Character

4.3.2 Block

4.3.3 Network

4.3.4 Line Discipline

4.4 Bus Type Idiosyncrasies

4.4.1 ISA

4.4.2 EISA

4.4.3 PCI

4.4.4 SCSI

4.4.5 PCCARD

5. Kernel Support

5.1 Data Structures

5.1.1 struct kern_devconf Structure

This structure contains some information about the state of the device and driver. It is defined in `/usr/src/sys/sys/devconf.h` as:

```
struct devconf {
    char dc_name[MAXDEVNAME];           /* name */
    char dc_descr[MAXDEVDESCR];        /* description */
    int dc_unit;                        /* unit number */
    int dc_number;                      /* unique id */
    char dc_pname[MAXDEVNAME];         /* name of the parent device */
    int dc_punit;                      /* unit number of the parent */
    int dc_pnumber;                    /* unique id of the parent */
    struct machdep_devconf dc_md;      /* machine-dependent stuff */
    enum dc_state dc_state;             /* state of the device (see above) */
    enum dc_class dc_class;            /* type of device (see above) */
    size_t dc_dataalen;                /* length of data */
    char dc_data[1];                   /* variable-length data */
};
```

5.1.2 struct proc Structure

This structure contains all the information about a process. It is defined in `/usr/src/sys/sys/proc.h`:

```
/*
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes. The process structure and the substructures
 * are always addressable except for those marked "(PROC ONLY)" below,
 * which might be addressable only on a processor on which the process
 * is running.
 */
struct proc {
    struct proc *p_forw;                /* Doubly-linked run/sleep queue. */
    struct proc *p_back;
    struct proc *p_next;                /* Linked list of active procs */
    struct proc **p_prev;               /* and zombies. */

    /* substructures: */
    struct pcred *p_cred;               /* Process owner's identity. */
    struct filedesc *p_fd;              /* Ptr to open files structure. */
    struct pstats *p_stats;             /* Accounting/statistics (PROC ONLY). */
    struct vmSPACE *p_vmSPACE;         /* Address space. */
    struct sigacts *p_sigacts;         /* Signal actions, state (PROC ONLY). */

#define p_ucred          p_cred->pc_ucred
#define p_rlimit        p_limit->pl_rlimit

    int    p_flag;                      /* P_* flags. */
    char   p_stat;                      /* S* process status. */
    char   p_pad1[3];

    pid_t  p_pid;                       /* Process identifier. */
    struct proc *p_hash;                 /* Hashed based on p_pid for kill+exit+... */
    struct proc *p_pgrpnext;            /* Pointer to next process in process group. */
    struct proc *p_pptr;                /* Pointer to process structure of parent. */
    struct proc *p_osptr;                /* Pointer to older sibling processes. */

    /* The following fields are all zeroed upon creation in fork. */
#define p_startzero      p_yspstr
    struct proc *p_yspstr;              /* Pointer to younger siblings. */
    struct proc *p_cpstr;                /* Pointer to youngest living child. */
};
```

```

pid_t   p_oppid;           /* Save parent pid during ptrace. XXX */
int     p_dupfd;          /* Sideways return value from fdopen. XXX */

/* scheduling */
u_int   p_estcpu;         /* Time averaged value of p_cpticks. */
int     p_cpticks;        /* Ticks of cpu time. */
fixpt_t p_pctcpu;         /* %cpu for this process during p_swtime */
void    *p_wchan;         /* Sleep address. */
char    *p_wmesg;         /* Reason for sleep. */
u_int   p_swtime;         /* Time swapped in or out. */
u_int   p_slptime;        /* Time since last blocked. */

struct  itimerval p_realtimer; /* Alarm timer. */
struct  timeval p_rtime;      /* Real time. */
u_quad_t p_uticks;          /* Statclock hits in user mode. */
u_quad_t p_sticks;          /* Statclock hits in system mode. */
u_quad_t p_iticks;          /* Statclock hits processing intr. */

int     p_traceflag;        /* Kernel trace points. */
struct  vnode *p_tracep;     /* Trace to vnode. */

int     p_siglist;          /* Signals arrived but not delivered. */

struct  vnode *p_textvp;     /* Vnode of executable. */

char    p_lock;             /* Process lock (prevent swap) count. */
char    p_pad2[3];          /* alignment */

/* End area that is zeroed on creation. */
#define p_endzero          p_startcopy

/* The following fields are all copied upon creation in fork. */
#define p_startcopy        p_sigmask

sigset_t p_sigmask;         /* Current signal mask. */
sigset_t p_sigignore;       /* Signals being ignored. */
sigset_t p_sigcatch;        /* Signals being caught by user. */

u_char  p_priority;         /* Process priority. */
u_char  p_usrpri;           /* User-priority based on p_cpu and p_nice. */
char    p_nice;             /* Process "nice" value. */
char    p_comm[MAXCOMLEN+1];

struct  pgrp *p_pgrp;       /* Pointer to process group. */

struct  sysentvec *p_sysent; /* System call dispatch information. */

struct  rtprio p_rtprio;     /* Realtime priority. */
/* End area that is copied on creation. */
#define p_endcopy          p_addr
struct  user *p_addr;        /* Kernel virtual addr of u-area (PROC ONLY). */
struct  mdproc p_md;         /* Any machine-dependent fields. */

u_short p_xstat;            /* Exit status for wait; also stop signal. */
u_short p_acflag;           /* Accounting flags. */
struct  rusage *p_ru;        /* Exit information. XXX */
};

```

5.1.3 struct buf Structure

The `struct buf` structure is used to interface with the buffer cache. It is defined in `/usr/src/sys/sys/buf.h`:

```

/*
 * The buffer header describes an I/O operation in the kernel.
 */
struct buf {
    LIST_ENTRY(buf) b_hash;           /* Hash chain. */
    LIST_ENTRY(buf) b_vnbufs;        /* Buffer's associated vnode. */
    TAILQ_ENTRY(buf) b_freelist;     /* Free list position if not active. */
    struct buf *b_actf, **b_actb;    /* Device driver queue when active. */
    struct proc *b_proc;             /* Associated proc; NULL if kernel. */
    volatile long b_flags;           /* B_* flags. */
    int b_qindex;                    /* buffer queue index */
    int b_error;                     /* Errno value. */
    long b_bufsize;                  /* Allocated buffer size. */
    long b_bcount;                   /* Valid bytes in buffer. */
    long b_resid;                    /* Remaining I/O. */
    dev_t b_dev;                     /* Device associated with buffer. */
    struct {
        caddr_t b_addr;              /* Memory, superblocks, indirect etc. */
    } b_un;
    void *b_saveaddr;                /* Original b_addr for physio. */
    daddr_t b_lblkno;                /* Logical block number. */
    daddr_t b_blkno;                 /* Underlying physical block number. */
    void (*b_iodone) __P((struct buf *)); /* Function to call upon completion. */
    void (*b_iodone) __P((struct buf *)); /* For nested b_iodone's. */
    struct iodone_chain *b_iodone_chain;
    struct vnode *b_vp;              /* Device vnode. */
    int b_pfcent;                    /* Center page when swapping cluster. */
    int b_dirtyoff;                  /* Offset in buffer of dirty region. */
    int b_dirtyend;                  /* Offset of end of dirty region. */
    struct ucred *b_rcred;           /* Read credentials reference. */
    struct ucred *b_wcred;           /* Write credentials reference. */
    int b_validoff;                  /* Offset in buffer of valid region. */
    int b_validend;                  /* Offset of end of valid region. */
    daddr_t b_pblkno;                /* physical block number */
    caddr_t b_savekva;               /* saved kva for transfer while bouncing */
};

void *b_driver1;                    /* for private use by the driver */
void *b_driver2;                    /* for private use by the driver */
void *b_spc;
struct vm_page *b_pages[(MAXPHYS + PAGE_SIZE - 1)/PAGE_SIZE];
int b_npages;
};

```

5.1.4 struct uio Structure

This structure is used for moving data between the kernel and user spaces through read() and write() system calls. It is defined in /usr/src/sys/sys/uio.h:

```

struct uio {
    struct iovec *uio_iov;
    int uio_iovcnt;
    off_t uio_offset;
    int uio_resid;
    enum uio_seg uio_segflg;
    enum uio_rw uio_rw;
    struct proc *uio_procp;
};

```

5.2 Functions

lots of 'em"

6. References.

FreeBSD Kernel Sources <http://www.freebsd.org>

NetBSD Kernel Sources <http://www.netbsd.org>

Writing Device Drivers: Tutorial and Reference; Tim Burke, Mark A. Parenti, Al, Wojtas; Digital Press, ISBN 1-55558-141-2.

Writing A Unix Device Driver; Janet I. Egan, Thomas J. Teixeira; John Wiley & Sons, ISBN 0-471-62859-X.

Writing Device Drivers for SCO Unix; Peter Kettle;

CONTENTS

1. Overview	1
2. Types of drivers.	1
2.1 Character	1
2.2 Block	4
2.3 Network	5
2.4 Line Discipline	5
3. Supported Busses	5
3.1 ISA -- Industry Standard Architecture	5
3.2 EISA -- Extended Industry Standard Architecture	7
3.3 PCI -- Peripheral Computer Interconnect	7
3.4 SCSI -- Small Computer Systems Interface	7
3.5 PCCARD (PCMCIA)	8
4. Linking Into the Kernel.	8
4.1 Standard Model	9
4.2 Loadable Kernel Module (LKM)	11
4.3 Device Type Idiosyncrasies	16
4.4 Bus Type Idiosyncrasies	16
5. Kernel Support	16
5.1 Data Structures	16
5.2 Functions	19
6. References.	20