6

Designing Loadable Kernel Servers

This chapter provides the basic information required to design loadable kernel servers.   It describes the code that every server must have to be integrated into the kernel and to provide services to its clients.   It also discusses the functions that loadable kernel servers can call, and it gives tips to help you write correct code that makes debugging easier.

Besides this chapter, you should also read either Chapter 7, ªNeXTbus Device Drivers,º or Chapter 8, ªNetwork Modules,º for detailed information about the type of server you're writing.

Choosing an Interface for Your Server

Before you can start writing your server, you have to decide whether it needs a message-based interface or a UNIX-style interface.   Network modules such as network protocols primarily use a third kind of interface (called a *network interface*, which is described in Chapter 8).   However most network modules have additional interface functions that are provided through either a message-based or a UNIX-style interface.

It is recommended that you use the Mach Interface Generator (MiG) to write a message-based interface.   One advantage of using a message-based interface is that your server can stay unloaded until the moment it's needed.   Message-based servers, besides having more intuitive interfaces than UNIX-style servers, also have the advantage of MiG's network independence.   For example, a graphics device with a message-based server could easily be accessed from any computer on the network.   MiG and Mach messages are described in detail in Chapter 2, ªUsing Mach Messages.º

Unfortunately, some servers can't use message-based interfaces.   For example, servers that interact with the UNIX file systemÐsuch as disk driversÐneed to supply a UNIX-style interface with UNIX entry points (*xxx*_**open()**, *xxx*_**close()**, and so on).

Tips for Writing Server Code

When developing your server, you should start by designing your server's interface.   When you're ready to start writing code, use a skeleton or sample server as the framework, and then add functionality a little at a time.   You might want to put all the major interface functions in place, but just have each one print a message that says it's been called.   This approach will make debugging your server much easier than if you implement large amounts of code at once.

As you write the code for your server, keep the following in mind:

- Using **register** isn't necessary. It's easier and often better to let the compiler decide what to put into registers.
- You must use **volatile** for variables that refer to hardware addresses, or that can be modified by interrupt functions or other threads.
- Beware of hardware registers that have side effects when accessed, or that contain different information when you read them than when you write to them.
- Don't declare large variables in functions. Instead, if you need a large variable, declare a pointer to it and then dynamically allocate space with **kalloc()** or **kget()**. (Automatic variables are allocated on the kernel stack. Since the kernel stack is only 4 KB, large variables can easily cause stack overflow and result in system panics that aren't easily debugged.)
- Don't recursively call functions. (Like large automatic variables, recursion can cause stack overflow.)

Functions Your Server Can Call

Your loadable kernel server can use the functions described in Chapter 10, ªKernel Support Functions.º In addition, it can use almost all Mach kernel functions, which are listed in Chapter 4, ªMach Functions.º

However, loadable kernel servers can't use functions that are not defined in the Mach kernel. For example, a few Mach kernel functions don't work because they rely on data that's only available at user level. For example, **mach_error()** can't be used in loadable kernel servers because it prints to **stderr**, which the Mach kernel doesn't have access to.

Note: You can't use C Thread, Network Name Server, Bootstrap Server, or Kernel-Server Loader functions or macros in your server, because they aren't part of the Mach kernel.

Warning: Loadable kernel servers run outside of the kernel task, even though they use the kernel address map. Be sure to specify the correct task and map to Mach kernel functions.

Executing as the Result of an Interrupt

When a function in your server is called as the direct result of an interrupt, the computer stops all normal processing until the function exits. Even if your server doesn't handle hardware interrupts, it can run as the result of a software interrupt. Specifically, functions that run because they were scheduled by a call to **ns_timeout()** or **ns_abstimeout()** (as described in Chapter 10) are running as the result of a software interrupt.

A server that's executing as the result of an interrupt must take certain precautions:

- It must not sleep.
- It must not call any functions that might sleep.
- If it needs to allocate memory, it should use **kget()** (which can fail but never sleeps).
- It shouldn't perform any I/O, unless the I/O is guaranteed not to block.

For information on handling hardware interrupts, see Chapter 7, ªNeXTbus Device Drivers.º

Building In Debugging Code

This section discusses functions that can help you debug in two ways: by displaying information while your server runs and by checking assumptions in your server. The functions and macros discussed in this section are explained further in Chapter 10.

Displaying Debugging Information

To display debugging information while your server runs, you have two choices: **kern_serv_log()** and the kernel **printf()** function. You should use **kern_serv_log()** instead of **printf()** whenever possible, since **printf()** slows the system and affects the timing of your server. You should use **printf()** only for unusual events that you need to see as soon as they occur and for events that are likely to result in a system panic.

Using kern_serv_log()

kern_serv_log() logs a message that a user process can later pick up and print out. The main advantages of **kern_serv_log()** are its quickness and reliability, even when called from an interrupt handler. However, if the system panics before the user process can pick up a log message, that message will be lost.

Your server supplies to **kern_serv_log()** the string to be logged and the priority at which it should be logged. Higher numbers correspond to higher priorities, but the exact interpretation of priority numbers is up to you.

Messages logged using **kern_serv_log()** can be obtained using either the kernel-server log command, **kl_log**, or any other user-level program that calls the functions **kern_loader_log_level()** and **kern_loader_get_log()**. By default, logging is off; you must set the log level before you can obtain any log messages. The **kl_log** utility is described in Appendix A, ªUtilities for Loadable Kernel Serversº; **kern_loader_log_level()** and **kern_loader_get_log()** are described in Chapter 3, ªUsing Loadable Kernel Servers.º

Using printf()

The kernel **printf()** function has the advantage that you can easily view its output. All you have to do is keep the console window open. However, **printf()** can greatly slow the system because nothing except hardware interrupt handling can happen during a call to **printf()**. Although **printf()** doesn't sleep, it's unreliable when called from an interrupt handler because messages can be garbled.

The kernel **printf()** function is best used when you have a short message that you want to see as soon as it happens. You can see **printf()** messages not only in the console window, but also in **/usr/adm/messages** and from a **msg** command in the NMI mini-monitor or Panic window. Your message is guaranteed to make it to the **msg** buffer (although it might be garbled) even if the kernel panics.

Checking Assumptions

To check assumptions in your server, you can use **ASSERT()** and **probe_rb()**.

Using ASSERT()

ASSERT() evaluates the expression you pass to it. If the result of the expression is 0, **ASSERT()** prints a message describing the line and file that the assertion failed on, and then calls **panic()**.

Note: **ASSERT()** doesn't do anything unless your server is compiled with the DEBUG C preprocessor macro defined. Other recommended compile flags are discussed in Chapter 9, ªBuilding, Loading, and Debugging Loadable Kernel Servers.º

Using probe_rb()

Use **probe_rb()** whenever you need to make sure that an address is valid. For example, to check whether a NeXTbus board is in a certain slot, you can call **probe_rb()**, passing the virtual address of one of the board registers.

Kernel-Server Loader Requirements

Your server must supply an instance variable to the kernel-server loader, **kern_loader**. You can also supply functions that **kern_loader** will call under certain circumstances, such as server initialization or shutdown.

You inform **kern_loader** of the instance variable name and of any functions to be called when you compile your server. This information goes into sections of your server's object file. See Appendix A for details on specifying information during compile time. The following sections first describe how to declare the instance variable in your code and then describe the types of functions you can write for **kern_loader** to call.

Instance Variable

Your server's instance variable is an ordinary C variable of type **kern_server_t** (defined in the header file **kernserv/kern_server_types.h**) that **kern_loader** uses to keep track of your server.

You can make the instance variable contain other information as well. Do this by defining a structure that begins with a field of type **kern_server_t**, followed by fields of your choice. For example, you might declare your instance variable in a header file as follows:

```
#import <kernserv/kern_server_types.h>

typedef struct my_instance_var
{
    kern_server_t  kern_server;  /* generic instance info */
    struct         my_dev;       /* per-device info */
    {
        int  field1;
        int  field2;
    } dev[MAX_MINE]
} my_instance_var_t;

my_instance_var_t  instance;
```

Writing Functions for kern_loader to Call

Your server can supply the following kinds of functions to **kern_loader**:

Kind of Function	Called When
Initialization	Your server is loaded.
Shutdown	Your server is unloaded.

Port server	Your server receives a message on a certain port.
Port death	A port for which your server has send rights dies.

Some servers might not require all or any of these functions. However, a server that doesn't start until it receives a message must supply **kern_loader** with the names and port servers of all ports that the server might receive its first message on.

Initialization functions can't be debugged with GDB. (You can't set a breakpoint in your server until it's fully loaded, and initialization functions are executed before then.) One way to get around this debugging problem is to have a message-based interface for initialization until your server is debugged. You can write a simple port server that initializes the server whenever it receives a message. You also have to write a simple program that sends this message. After you've finished debugging the initialization sequence, you can move it into the initialization functions called by **kern_loader**.

Shutdown functions are often used to free kernel resources. When the server is unloaded, no other part of the kernel can contain a reference to any code or data contained within the loadable server. If the kernel tries to refer to any code or data in an unloaded server, the system panics.

Considerations for Message-Based Servers

Message-based loadable kernel servers are built using the Mach Interface Generator (MiG). Examples of using MiG are in Chapter 2 of this manual, as well as under **/NextLibrary/Documentation/NextDev/Examples**. Under the **Examples** directory, the **MiG** example contains only user-level code, but it's a good starting place if you're unfamiliar with MiG. The **Log** example features a loadable kernel server that can receive only one type of request. The **ServerVsHandler** example has two versions of a loadable kernel server that can receive two types of requests.

Server versus Handler Interfaces

A message-based loadable kernel server can have one of two interfaces: a server interface or a handler interface. MiG automatically produces a server interface, but if requested it can produce most of the code for a handler interface. You have to write just a little more code to complete a MiG-generated handler interface. Handler interfaces have a performance advantage because they can use much less wired kernel memory if the loadable kernel server never returns much data.

For example, for a loadable kernel server with a server interface, MiG allocates reply messages that are MAX_SIZE_BYTES (currently 8192) bytes long. If the server returns only a little dataÐfor example, if it has ten interface functions, each returning only an integer indicating the call's successÐthen a lot of kernel memory is being wasted. If the same server has a handler interface, MiG allocates only 32 bytes for each reply message (24 bytes for the header, and 8 bytes for the message body).

The difference in memory usage happens because MiG always generates large messages to return server data, but for handlers, it allocates just enough space to hold the largest message that's returned.

Note: From the caller's point of view and from the point of view of the loadable kernel server function that the MiG-generated interface calls, handler and server interfaces look identical. The difference between them is visible only in the options you specify to MiG, the files that MiG generates, and the extra bit of code you must write, as described in the following procedure.

To convert a server to a handler, while keeping the exact same interface to user processes, follow these steps (substituting the name of your server for *name* and ªmydriverº):

1. Copy your loadable kernel server's ª.defsº file to another directory and enter the following command at a UNIX shell prompt: **mig** *name***.defs -handler** *name***Handler.c -sheader** *name***Handler.h -user** *name***User.c -header** *name***.h**. For example:

   ```
   mymachine> mig mydriver.defs -handler mydriverHandler.c -sheader
   mydriverHandler.h -user mydriverUser.c -header mydriver.h
   ```

2. This creates a group of files. Look in the *name***Handler.h** file for the *name***_t** structure. Here's an example of what it looks like:

   ```
   typedef struct mydriver {
       void    *arg;      /* argument to pass to function */
       int      timeout;  /* timeout for RPC return msg_send */

       /* Routine mydriver_do_log */
       kern_return_t (*mydriver_do_log) (
       void *server);
   } mydriver_t;
   ```

3. Import *name***Handler.h** to your server and add a global variable of type *name*_t that maps user functions to the corresponding function in your server. The kernel and the kernel loader will use this structure to call your server's functions. For example, for a ª.defsº file that has the line ªroutine mydriver_do_log(server: port_t);º, you might have the following:

   ```
     #import "mydriverHandler.h"
     kern_return_t mydriver_do_log(port_t server);

     mydriver_t mydriver_funcs = {
         0,
         100,            /* in milliseconds */
         (kern_return_t (*)(void *))mydriver_do_log
     };
   ```

4. In the load commands script, change all instances of SMAP to HMAP, and change each function argument to the name of the mapping structure. For example:

   ```
     SMAP          mydriver0 mydriver_server 0
   ```

   becomes:

   ```
     HMAP          mydriver0 mydriver_handler mydriver_funcs
   ```

5. Change your server's makefile so that it performs the MiG command described above and has the correct dependencies. For example:

   ```
     MIGOUTPUT=mydriverUser.c mydriverServer.c mydriver.h
     SERVER_OBJ= mydriver_main.o mydriverServer.o
     .
     .
     .
     mydriver_reloc: ${SERVER_OBJ} LoadCommands UnloadCommands
         kl_ld -n mydriver -l LoadCommands -u UnloadCommands -i instance \
         -d mydriver_loadable -o $@ ${SERVER_OBJ}

     ${MIGOUTPUT}: mydriver.defs
         mig mydriver.defs
   ```

   might become:

   ```
     MIGOUTPUT=mydriverUser.c mydriverHandler.c mydriverHandler.h \
               mydriver.h
     SERVER_OBJ= mydriver_main.o mydriverHandler.o
     .
     .
     .
     mydriver_reloc: ${SERVER_OBJ} LoadCommands UnloadCommands
   ```

```
     kl_ld -n mydriver -l LoadCommands -u UnloadCommands -i instance \
    -d mydriver_loadable -o $@ ${SERVER_OBJ}

${MIGOUTPUT}: mydriver.defs
    mig mydriver.defs -handler mydriverHandler.c \
    -sheader mydriverHandler.h -user mydriverUser.c \
    -header mydriver.h

${SERVER_OBJ}:    mydriverHandler.h
```

For another example of converting a server to a handler, see the files under **/NextLibrary/Documentation/NextDev/Examples/ServerVsHandler**.

Sending and Receiving Out-of-Line Data

A message-based server that receives out-of-line data doesn't have direct access to the data. The data is inaccessible because it appears in the address map of the server's task, but loadable kernel servers use the kernel's address map instead of their own task's map. To read out-of-line data, your server must use **vm_write()** to map all or part of the data into the kernel map. Similarly, to send out-of-line data, your server must first call **vm_read()** to map the data into the address map of the server's task. Both **vm_write()** and **vm_read()** work on entire pages, so the data sent out-of-line must be page-aligned. However, as long as none of the data is wired down, the data isn't copied until it's written to.

Note: Don't use **copyin()** and **copyout()** in message-based servers. They work only in UNIX-style servers.

Considerations for UNIX-Based Servers

For a UNIX-based server, you must provide the proper entry points and insert the server into the appropriate device switch tables. This section lists the entry points you need, but doesn't cover most entry points in detail. If an entry point is not sufficiently covered here, see Egan and Teixeira's *Writing a UNIX Device Driver*.

Note: You should call NeXT Technical Support to receive the device major number to use. Getting the major number from NeXT will help ensure that your server works with other NeXT-supplied and third-party servers.

UNIX Device Entry Points

This section shows all the entry points that a character or block driver can provide.

Character-Device Entry Points

Entry points for character devices are defined in the header file **sys/conf.h** as shown below.

```
struct cdevsw
{
    int  (*d_open)();
    int  (*d_close)();
    int  (*d_read)();
    int  (*d_write)();
    int  (*d_ioctl)();
    int  (*d_stop)();
    int  (*d_reset)();
    int  (*d_select)();
    int  (*d_mmap)();
    int  (*d_getc)();
    int  (*d_putc)();
};
extern struct  cdevsw cdevsw[];
```

Field	Description
d_open	A pointer to the server function that handles an **open()** system call.
d_close	A pointer to the server function that handles a **close()** system call.
d_read	A pointer to the server function that handles a **read()** system call.
d_write	A pointer to the server function that handles a **write()** system call.
d_ioctl	A pointer to the server function that handles an **ioctl()** system call.
d_stop	Not supported for user-written servers.
d_reset	Not used.
d_select	A pointer to the server function that handles a **select()** system call. If your device is ready for reading or writing, this function should return true. If your device is *always* ready for reading and writing, you can specify **seltrue()** in the **cdevsw** table, which will make the kernel return true without calling your server.
d_mmap	A pointer to the server function that handles memory mapping of device space to user space. This function, which is typically found in frame buffers, must return the page number of the passed offset.
d_getc	Not supported for user-written servers; used for console devices.
d_putc	Not supported for user-written servers; used for console devices.

If your driver implements the *d_select* entry point, it usually handles the case that no data is immediately available. In this case, your driver must remember that someone is interested in the data, as well as whether more than one party is interested. In typical UNIX systems, a pointer to the **proc** structure of the interested party is saved. In NeXT systems, however, the driver must save a pointer to a thread. The **selthreadcache()** function stores this thread pointer and ensures that the thread won't go away before **selthreadclear()** is called. The **selthreadclear()** function should be called when the awaited device activity has taken place or when the device is closed. The following example illustrates how to use **selthreadcache()** and **selthreadclear()**.

```
typedef struct {
    ...
    void    *selread;
    void    *selwrite;
    void    *selexcep;
    int      collision : 1;
    ...
} Mystruct;

xyzselect(dev, flag)
{
    Mystruct *myinfo = Myinfo[minor(dev)];

   switch (flag) {
    case FREAD:
        if ( data not available ) {
            if ( selthreadcache(&myinfo->selread) )
                // more than one party...
                myinfo->collision = 1;
        }
        ...
        break;
    ...
    }
    ...
}

xyzint(dev)
{
    ...
    if (  read data && myinfo->selread )
        xyzwakeup(myinfo, FREAD);
    ...
}

xyzwakeup(myinfo, flag)
Mystruct *myinfo;
{
    if (flag & FREAD) {
        int oldpri = splxyz();    // may not be necessary

       if (myinfo->selread) {
            selwakeup(myinfo->selread, myinfo->collision);
            selthreadclear(&myinfo->selread);
            myinfo->collision = 0;
        }
        splx(oldpri);
    }
    ...
}

xyzclose()
{
    ...
    if (myinfo->selread)
        selthreadclear(&myinfo->selread);
    ...
}
```

Block-Device Entry Points

Entry points for block devices are defined in the header file **sys/conf.h** as shown below.

```
struct bdevsw
{
    int  (*d_open)();
    int  (*d_close)();
    int  (*d_strategy)();
    int  (*d_dump)();
    int  (*d_psize)();
    int  d_flags;
};
extern struct  bdevsw bdevsw[];
```

Field	Description
d_open	A pointer to the server function that handles an **open()** system call.
d_close	A pointer to the server function that handles a **close()** system call.
d_strategy	A pointer to the server function that eventually handles **read()** and **write()** system calls.
d_dump	A pointer to the server function that dumps physical memory to the swap device when the system is going down. Used only for devices that can be

used for swapping.

d_psize	A pointer to the server function that returns the size of the swap partition for swap devices. Used only for devices that can be used for swapping.
d_flags	Contains flags that give more information about the device to the kernel. The only defined flag is B_TAPE, which tells the kernel that it can't reorder I/O to this server.

Inserting UNIX Servers into Device Switch Tables

If your server is entered through UNIX system calls, you must insert it into the appropriate device switch tables during your server's initialization. While debugging, you should do this through a message-based interface. Later, you can transfer this to an initialization function called by **kern_loader**.

The following example illustrates a server inserting itself into switch tables. Since the example is taken from a block driver, the server inserts itself into both the **bdevsw** and **cdevsw** tables. Character drivers have to insert themselves only into the **cdevsw** table.

Note: In the following example, MY_BLOCK_MAJOR and MY_RAW_MAJOR are device major numbers, which you should obtain from NeXT Technical Support.

```
/*
 * Example of a driver inserting itself into the block and character
 * device switch tables.
 */

#import <sys/conf.h>

extern int nulldev();
extern int nodev();
extern int seltrue();
#define nullstr 0

struct bdevsw my_bdevsw =  {
                 int (*myopen)(),
                 int (*myclose)(),
                 int (*mystrategy)(),
                 nodev,
                 nodev,
                 0 };
struct cdevsw my_cdevsw =  {
                 int (*myopen)(),
                 int (*myclose)(),
                 int (*myread)(),
                 int (*mywrite)(),
                 int (*myioctl)(),
                 nodev,
                 nulldev,
                 seltrue,
                 nodev,
                 nodev,
                 nodev };

struct bdevsw my_saved_bdevsw;
struct cdevsw my_saved_cdevsw;

/* Save whatever entries were in the tables for our major numbers. */
my_saved_bdevsw = bdevsw[MY_BLOCK_MAJOR];
my_saved_cdevsw = cdevsw[MY_RAW_MAJOR];
```

```
/* Put my entries in the switch tables. */
bdevsw[MY_BLOCK_MAJOR]= my_bdevsw;
cdevsw[MY_RAW_MAJOR]  = my_cdevsw;
```