Writing Loadable Kernel Servers

NeXT Developer's Library

NeXTstep

Draw upon the library of software contained in NeXTstep to develop your applications. Integral to this development environment are the Application Kit and Display PostScript.

Concepts

A presentation of the principles that define NeXTstep, including user interface design, object-oriented programming, event handling, and other fundamentals.

Reference, Volumes 1 and 2

Detailed, comprehensive descriptions of the NeXTstep Application Kit software.

Sound, Music, and Signal Processing

Let your application listen, talk, and sing by using the Sound Kit and the Music Kit. Behind these capabilities is the DSP56001 digital signal processor. Independent of sound and music, scientific applications can take advantage of the speed of the DSP.

Concepts

An examination of the design of the sound and music software, including chapters on the use of the DSP for other, nonaudio uses.

Reference

Detailed, comprehensive descriptions of each piece of the sound, music, and DSP software.

NeXT Development Tools

A description of the tools used in developing a NeXT application, including the Edit application, the compiler and debugger, and some performance tools.

NeXT Operating System Software

A description of NeXT's operating system, Mach. In addition, other low-level software is discussed.

Writing Loadable Kernel Servers

How to write loadable kernel servers, such as device drivers and network protocols.

NeXT Technical Summaries

Brief summaries of reference information related to NeXTstep, sound, music, and Mach, plus a glossary and indexes.

Supplemental Documentation

Information about PostScript, RTF, and other file formats useful to application developers.

Writing Loadable Kernel Servers

Manual written by Kathy Walrath
Edited by Caroline Rose and Helen Casabona
Book design by Eddie Lee
Illustrations by Jeff Yaksick and Don Donoughe
Production by Adrienne Wong, Jennifer Yu, and Katherine Arthurs
Publications management by Cathy Novak

Reorder Product #N6007B

Contents

Introduction

This manual describes how to write loadable kernel servers for NeXT™ computers. It's part of a collection of manuals called the *NeXT Developer's Library*; the illustration on the first page of this manual shows the complete set of manuals in this Library.

This manual describes how to write loadable kernel servers for NeXT computers. Loadable kernel servers are the only way for third-party developers to add functionality (such as device drivers) to the NeXT Mach kernel.

To write a loadable kernel server, you must be able to program in C. Depending on the kind of loadable kernel server you're writing, you probably don't need to be familiar with UNIX® internals, but a general background in writing interrupt-driven device drivers or networking protocols is useful.

Some topics aren't covered here in detail; instead, you're referred to a generally available book on the subject, or to an on-line source of the information. For example, Egan and Teixeira's *Writing a UNIX Device Driver* describes writing UNIX-style drivers and gives general driver-writing tips; those subjects aren't described in detail in this manual.

A version of this manual is stored on-line in the NeXT Digital Library (which is described in the user's manual *NeXT Applications*). The Digital Library also contains Release Notes that provide last-minute information about the latest release of the software.

How This Manual is Organized

The first four chapters of this manual describe different aspects of developing loadable kernel servers, with Chapter 4 concentrating on network-related kernel servers. Chapter 5 provides detailed descriptions of each C function you can use inside a loadable kernel server; these functions are summarized in Appendix F. Appendices A through E explain how to use several programs, such as the KGDB kernel debugger.

Conventions

Syntax Notation

Where this manual shows the syntax of a function, command, or other programming element, the use of bold, italic, square brackets [ ], and ellipsis has special significance, as described here.

Bold denotes words or characters that are to be taken literally (typed as they appear). *Italic* denotes words that represent something else or can be varied. For example, the syntax

print *expression*

means that you follow the word **print** with an expression.

Square brackets [ ] mean that the enclosed syntax is optional, except when they're bold **[ ]**, in which case they're to be taken literally. The exceptions are few and will be clear from the context. For example,

pointer [*filename*]

means that you type a pointer with or without a file name after it, but

[*receiver message***]**

means that you specify a receiver and a message enclosed in square brackets.

Ellipsis (...) indicates that the previous syntax element may be repeated. For example:

Syntax	**Allows**
pointer ...	One or more pointers
pointer [, *pointer*] ...	One or more pointers separated by commas
pointer [*filename* ...]	A pointer optionally followed by one or more file names
pointer [, *filename*] ...	A pointer optionally followed by a comma and one or more file names separated by commas

Special Characters

In general, notation like

Alternate-x

represents the character you get when you hold down the Alternate key while typing x. Because the modifier keys Alternate, Command, and Control interpret the case of letters differently, their notation is somewhat different:

Notation	**Meaning**
Alternate-x	Hold down Alternate while typing lowercase x.
Alternate-X	Hold down Alternate while typing uppercase X (with either Shift or Alpha Lock).
Alternate-Shift-x	Same as Alternate-X.
Command-d	Hold down Command while typing lowercase d; if Alpha Lock is on, pressing the D key will still produce lowercase d when Command is down.
Command-Shift-D	Hold down Command and Shift while pressing the D key. Alpha Lock won't work for producing uppercase D in this case.
Control-X	Hold down Control while pressing the X key, with or without Shift or Alpha Lock (case doesn't matter with Control).

Notes and Warnings

Note: Paragraphs like this contain incidental information that may be of interest to curious readers but can safely be skipped.

Warning: Paragraphs like this are extremely important to read.

Chapter 1
Overview

This manual describes how to write loadable kernel servers for NeXT computers. A loadable kernel server is any code that's added to the kernel after the system has been booted. Currently, the only types of loadable kernel servers that third party developers can write are network protocols, packet sniffers, and device drivers for NeXTbus™ boards.

Note: It currently isn't possible for third parties to write drivers for any on-board interfaces, such as SCSI.

Before You Start

Decide whether you really need to write a loadable kernel server. For a NeXTbus board where response time isn't crucial, you can avoid writing a server by writing a user-level program that calls the slot driver. The slot driver is a sample server in the NeXTbus Development Kit (NeXT product number N7002).

Note: You can't currently use DMA (direct memory access) in a user-written driver. This is because all the CPU board DMA channels are used by existing drivers, and DMA isn't currently supported with NeXTbus boards. Instead, you must use programmed (direct) I/O.

Get the Documentation You Need

Depending on the type of server you're writing, you'll need some or all of the documentation listed in this section. You'll also need the server examples that are under **/NextLibrary/Documentation/NextDev/Examples/ServerVsHandler** and in the NeXTbus Development Kit.

You'll definitely need the manuals *NeXT Operating System Software* and *NeXT Development Tools*, which are part of the *NeXT Developer's Library.*

You might need to have the *Network and System Administration* manual on hand, especially when you're setting up your computers. This manual comes with every NeXT computer.

If you're writing a driver for a NeXTbus board, you'll need the hardware specifications and installation guide. These are all part of the NeXTbus Development Kit.

- *NeXTbus Interface Chip Specification*
- *NeXTbus Specification*
- *NeXTbus Development Kit Installation Guide*

The following book has good information for any driver writer. It also has specific information on UNIX drivers.

Egan, Janet I., and Teixeira, Thomas J., *Writing a UNIX Device Driver.* New York: John Wiley & Sons, Inc., 1988.

If you're writing a server with a UNIX-style interface, you might need the following book's information on the UNIX 4.3BSD operating system.

Leffler, Samuel J., McKusick, Marshall Kirk, Karels, Michael J., and Quarterman, John S., *The Design and Implementation of the 4.3BSD UNIX Operating System.* New York: Addison-Wesley Publishing Company, 1988.

Get the Hardware You Need

To write a server, you'll need the following equipment:

- Two NeXT computers: a development computer and a test computer. (The server you're creating will run on the test computer.)

- Either a working network connection for both computers or a cable to connect their serial ports. If you don't have a network, and the computers have two different CPUs (one is an 68040 and one is an 68030), then the serial cable must be customized. The network connection and cable are described in detail in Appendix D, "The Kernel Debugger."

- A NeXTbus Interface Chip (NBIC) on the CPU board of the test computer (only if you're writing a driver for a NeXTbus device). The NBIC is already installed in every NeXTcube™ computer and 68040 Upgrade Board.

- The hardware (if any) your server will control, installed in or connected to the test machine.

The test computer should use as little disk space as possible; the more disk space it uses, the more time it will take to reboot after a system panic. You can avoid using local disk space by making this computer a NetBoot client. See Appendix D for more information on setting up the two computers.

Choose 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.

We recommend 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 isn't loaded 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.

However, some servers require other interfaces. For example, servers that interact with the UNIX file system—such as disk drivers—need to supply a UNIX-style interface with UNIX entry points (*xxx*_**open**(), *xxx*_**close**(), and so on).

If you write a UNIX-style server, you must call NeXT Technical Support to receive the device major number you must use. Getting the major number from NeXT will help ensure that your server works with other NeXT-supplied and third-party servers.

Software Support

Several software programs can help you write and debug your server:

- The NeXTbus Probe, an application in the NeXTbus Development Kit, lets you access registers on NeXTbus boards.
- The Mach Interface Generator (MiG) helps generate your server's message-based interface.
- The kernel-server utility (**kl_util**) lets you load and unload your server from the kernel.
- The kernel-server log command (**kl_log**) lets you obtain messages logged by your server.
- The kernel debugger (KGDB) helps you debug your server.

The NeXTbus Probe

The NeXTbus Probe is a NeXTstep® application that lets you read and write to any address on a NeXTbus board. It's useful in all phases of driver writing. When you first start, it can help you verify the address of each register. As you write your server, you can test sequences of reads and writes with the NeXTbus Probe before you put the code into your server. Later, the NeXTbus Probe can help you when you're testing or debugging your server.

The Mach Interface Generator (MiG)

If you use MiG to write your server's interface, you don't have to use UNIX system calls as entry points. Instead, you can create an intuitive network-independent interface for your server. MiG also helps by creating the header file that user programs must include to send messages to your server. MiG is described in the *Operating System Software* manual.

The Kernel-Server Utility (kl_util)

You can use the **kl_util** command in a shell window to load and unload your server. You can also use **kl_util** to get the status of the kernel server loader (**kern_loader**), shut down **kern_loader**, and allocate or deallocate resources for a server within **kern_loader**. See Appendix B, "The Kernel-Server Utility," for instructions on using **kl_util**.

The Kernel-Server Log Command (kl_log)

By using the **kl_log** command in a shell window, you can get the messages that your server logs with the **kern_serv_log**() function. You can also specify how important a message must be to be kept. See Appendix C, "The Kernel-Server Log Command," for instructions on using **kl_log**.

The Kernel Debugger (KGDB)

KGDB is a superset of GDB, the GNU source-level debugger. KGDB includes the standard GDB commands, plus a few that are designed specifically for debugging kernels. You load the server you're debugging into the test computer's kernel and then use KGDB on the development computer to watch the server's behavior. See Appendix D for instructions.

Concepts

This section introduces the concepts you'll need to understand before you write your server.

Loadable Kernel Servers

A *kernel server* is a thread that runs in the address space of the Mach kernel and implements a server interface. Examples of kernel servers are the MIDI and Sound/DSP device drivers, as well as the Mach system call interface itself. PostScript®, the Network Name Server (**nmserver**), and the Pasteboard Server are examples of Mach servers not running within the kernel.

In general, a server should run in the kernel if it needs to respond quickly to hardware interrupts. If quick response time isn't necessary, the server can run outside of the kernel.

A *loadable* kernel server is a server that's loaded into the kernel after the system has been booted. This allows you to change the server without recompiling the kernel. A loadable kernel server can also be unloaded when it's no longer required, thus saving memory and other resources. Being able to unload and reload a server also speeds up development time because you don't have to reboot every time you change the server. The MIDI driver is an example of a loadable kernel server.

Loadable kernel servers have three states:

- Allocated. The kernel-server loader (**kern_loader**) has allocated space and resources for the loadable kernel server and is listening for messages to its ports. However, the server isn't currently loaded into the kernel.
- Loaded. The loadable kernel server is running. It runs as a thread in a non-kernel task, but with the kernel's address map.
- Unallocated. **kern_loader** has no space or other resources allocated for the loadable kernel server.

Loadable kernel servers can be built with or without Mach messages. A kernel server that's based on messages has the advantage that it isn't loaded until it's needed. This works because **kern_loader**, when it allocates resources for a kernel server, creates the kernel server's ports and advertises them with the Network Name Server. When a message is received on one of these advertised ports, **kern_loader** loads the allocated server into the kernel and forwards the advertised ports along with the received message to the now loaded and running kernel server.

Loadable kernel servers that don't use messages must be loaded into the system as soon as they're allocated in **kern_loader**. Like other loadable kernel servers, though, they can be allocated at any time and unloaded when they're no longer needed. Since newly loaded, non-message-based kernel servers aren't known to the kernel, they must add their entry points to the system. For example, a UNIX-style server must insert pointers to its entry points into device switch tables.

A kernel server doesn't have to use messages if it's accessed only by table lookups. For example, UNIX-style device drivers and networking protocols can be implemented without messages.

The Kernel-Server Loader

The kernel-server loader, **kern_loader**, is responsible for initializing and loading loadable kernel servers. When started, **kern_loader** reads a configuration file and allocates the listed kernel servers. Each server gives **kern_loader** its name and the names of its ports to be advertised with the Network Name Server. Also specified are initialization and shutdown sequences and the mapping of advertised ports to message-handler routines within the server. The initialization sequence can include information on whether the server should be *wired down* (made memory-resident in kernel virtual memory), and whether the kernel server should be loaded into the kernel immediately or wait for a received message.

Besides allocating kernel servers when **kern_loader** starts, you can use the kernel-server utility, **kl_util**, to allocate them while **kern_loader** is running. See Appendix B for more information on using **kl_util** for run-time control of **kern_loader**.

When **kern_loader** allocates a server, it allocates memory within the Mach kernel and relocates the file against the running kernel's symbol table at the allocated address. The result is a loadable object file containing code and data that **kern_loader** can directly copy into the kernel virtual address space. KGDB uses this loadable object file when you debug your server.

kern_loader receives messages on behalf of all allocated, but unloaded, kernel servers. When the first message is received for one of these servers, the server is loaded into the kernel. During the loading process, **kern_loader** not only loads the server but also initializes it. Initialization includes setting up port mappings and calling any server routines that were specified to be in the initialization sequence. The kernel server is also wired down if requested.

Once the kernel server is loaded and running, **kern_loader** doesn't interact with it unless some extraordinary condition arises. This may be due to a user request (for example, using **kl_util** to unload the server) or termination or reconfiguration of **kern_loader**.

When shutting down a running kernel server, **kern_loader** calls the routines in the server's shutdown sequence.

See Appendix A, "The Kernel-Server Loader," for more information on using **kern_loader**, including how to specify the configuration information for your server.

The Hardware

NeXTcube computers and the original NeXT Computer have four slots: slots 0, 2, 4, and 6. As Figure 1-1 shows, when you look at the back of the cube the slots are numbered (from left to right) 6, 2, 0, 4. The CPU board is always in slot 0. The remaining slots are available for other NeXTbus boards. All the boards plug into and communicate over the NeXTbus.

Note: NeXTstation™ computers don't have a NeXTbus or any expansion slots.

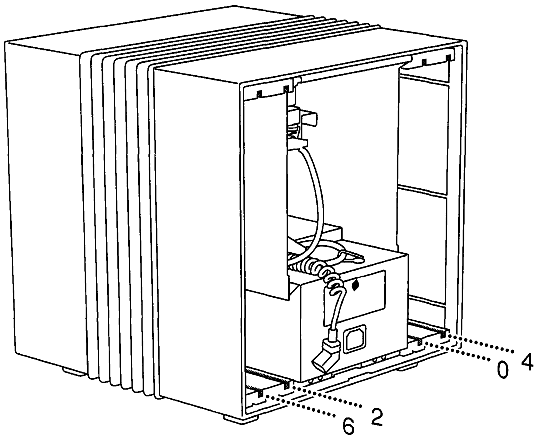

Figure 1-1. Slot Order

Before your server can talk to any NeXTbus boards, you must have a NeXTbus Interface Chip (NBIC) on your CPU board. See the *NeXTbus Development Kit Installation Guide* for instructions if you need to install an NBIC onto a 68030 board.

NeXTbus Address Space

Each NeXTbus board has access to two sections of the NeXTbus physical address space: its slot's *board address space* and its slot's *slot address space*. Each slot's board address space is 256 megabytes, from 0x*s*0000000 to 0x*s*fffffff, where *s* is the slot number; each slot's slot address space is 16 megabytes at addresses 0xf*s*000000 to 0xf*s*ffffff. Every register that your server must read or write will appear in either the board or the slot address space; the specification for your hardware should tell you exactly which physical address you must use.

Warning: Don't use addresses 0x*s*200c000 to 0x*s*200cfff, where *s* is 2, 4, or 6. Because these addresses are intercepted by logic on the CPU board, writing to them can cause errors. If your server needs to access information at these addresses, your board must be configured to accept addresses for the next higher slot; Chapter 2, "Designing Kernel Servers," describes how to do this.

Figure 1-2 shows the NeXTbus physical address space. As the figure shows, the NeXTbus architecture allows for up to 15 slots, each with up to 272 megabytes of physical address space.

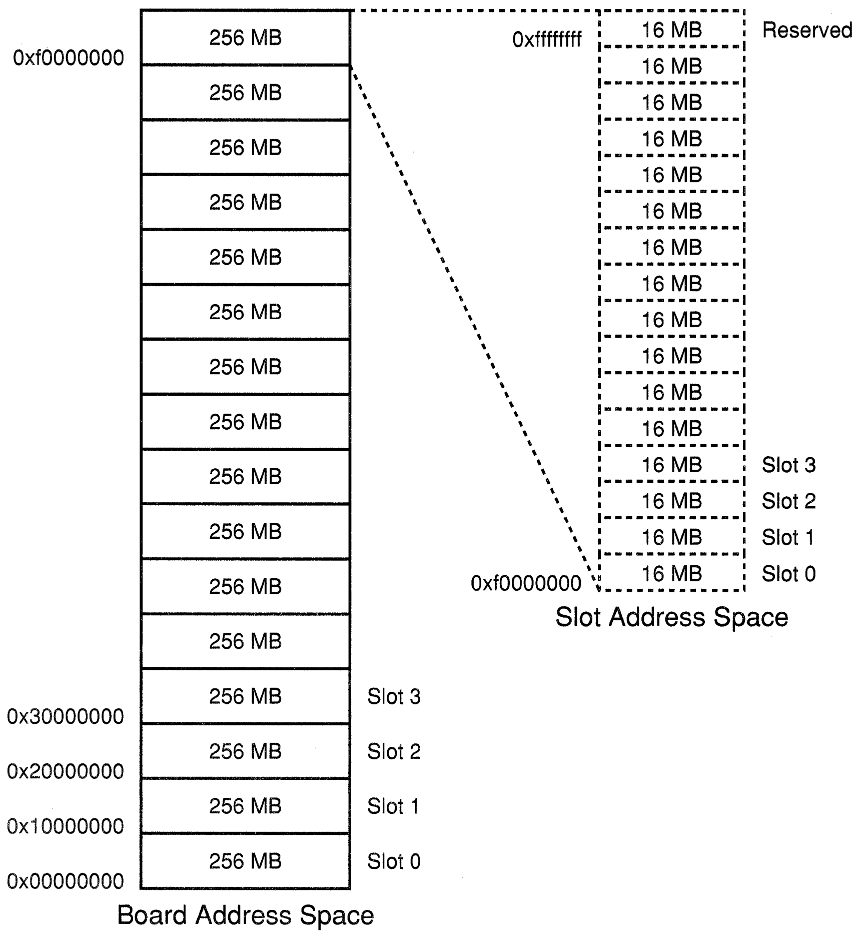

Figure 1-2. NeXTbus Address Space

To access a hardware address, your server must first map the physical address into a virtual one. Chapter 2 describes how to perform this mapping.

The NeXTbus Interface Chip (NBIC)

The NBIC is a chip that allows boards to talk to the NeXTbus. The CPU board talks to the NeXTbus with an NBIC. NeXTbus boards can also use the NBIC to talk to the NeXTbus, but they don't have to. Whether or not your NeXTbus board uses an NBIC, your board must still conform to the *NeXTbus Specification*, which includes having six bytes of ID and interrupt control information. These six bytes are discussed in Chapter 2, along with more information on the NBIC.

Chapter 2
Designing Kernel Servers

This chapter provides the basic information required to design your loadable kernel server.

Rules of Thumb

You should start by designing your server's interface. When you're ready to start coding, use a skeleton or sample server as the framework, and then add functionality a little at a time. You may want to put all the major interface routines 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.

When you write the code for your server, follow these rules:

- Don't use **register**. It's easier and often better to let the compiler decide what to put into registers.
- Use **volatile** for variables that refer to hardware addresses or that can be modified by interrupt routines 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.
- Define the C preprocessor macros KERNEL, KERNEL_FEATURES, and MACH every time you compile your server. For example:

  ```
  cc -g -DKERNEL -DKERNEL_FEATURES -DMACH -c myserver.c
  ```

- Until you've finished debugging your server, compile with debugging information (**-g**) so you can easily use the kernel debugger. You might also want to define the DEBUG C preprocessor macro if you use **ASSERT**() (described later).
- Don't compile with optimization (**-O**) until you've finished debugging your server. Optimization can make variables seem to have the wrong data when you check their values in the kernel debugger.
- 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, causing system panics for which the kernel debugger can't find a cause.)

- Don't recursively call functions. (Like large variables, recursion can cause stack overflow.)

Testing Your Hardware

The main tool for software-based testing of NeXTbus boards is the NeXTbus Probe. When you first start, try reading from and/or writing to every register on the board. This will help you verify the address and format of each register.

Before you access the NeXTbus board with your server, you can use the NeXTbus Probe to manually enter the actions that you think your server will have to take. Once you've made sure the actions work, you can put the code to do them into your server.

For non-NeXTbus drivers, use the ROM monitor to read and write physical addresses. It's described in Appendix E, "The ROM Monitor and NMI Mini-Monitor."

Building In Debugging Code

This section discusses routines that can help you debug in two ways: by displaying information while your server runs and by checking assumptions in your server. The routines and macros discussed in this section are explained further in Chapter 5, "C Functions."

Displaying Debugging Information

To display debugging information while your server runs, you have two choices: **kern_serv_log()** and the kernel **printf()** routine. 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.

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.

You supply to **kern_serv_log()** the string to be logged and the priority at which the string 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 reset the log level before you can obtain any log messages. **kl_log** is described in Appendix D, "The Kernel-Server Log Command"; **kern_loader_log_level()** and **kern_loader_get_log()** are described in the *Operating System Software* manual.

printf()

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

The kernel **printf()** routine 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()**.

ASSERT()

ASSERT() evaluates the expression you pass it. If the expression's result 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.

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's registers.

Kernel-Server Loader Requirements

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

You inform **kern_loader** of the instance variable's name and of any routines to be called when you compile your server. This information goes into sections of your server's object file. See Appendix A, "The Kernel-Server Loader," for information 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 routines you can write for **kern_loader** to call.

The 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 Routines for kern_loader to Call

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

Kind of Routine	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 routines. However, a server that doesn't start until it receives a message must supply **kern_loader** with the names of all ports that the server might receive its first message on.

Initialization routines can't be debugged with KGDB. (You can't set a breakpoint in your server until it's fully loaded, and initialization routines 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 (without using MiG, if your server doesn't use any other messages) that will initialize the server whenever it receives a message. You have to write a simple program that sends this message, and you have to specify to **kern_loader** that your server starts up as soon as it's loaded. After you've finished debugging the initialization sequence, you can move it into the initialization routines called by **kern_loader**.

Shutdown routines 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 reference any code or data in an unloaded server, the system panics.

Writing an Interrupt Handler

If your server must directly detect interrupts, you must provide an interrupt handler.

For NeXTbus drivers, this interrupt handler might be called whenever *any* NeXTbus board interrupts, not just when a board controlled by your server interrupts. This means that your interrupt handler must check whether your server's hardware generated the interrupt. If it did, your server must handle the interrupt, stop the hardware from generating this interrupt, and return true. If your server's hardware did not generate the interrupt, your interrupt handler must do nothing and return false.

To stop NeXTbus hardware from generating an interrupt, write a 1 to bit 7 of the Interrupt Mask byte. This byte is discussed later in this chapter.

Warning: Your interrupt handler (and any routine it might call) must not sleep.

Because interrupt handlers can't sleep, they can't allocate memory except by using **kget**(), which isn't guaranteed to succeed. They also can't perform any I/O unless it's guaranteed not to block.

Because interrupt handlers execute on behalf of the hardware, they have no knowledge of which user process they're working for. Thus, they can't access anything to do with a user process.

You install interrupt handlers using **install_polled_intr**(), and remove them using **uninstall_polled_intr**().

Below is an example of an interrupt handler.

```
#define SLOT_INTR_BIT 0x80
#define SLOTCOUNT 4
/* In the following macro, slotid is half the slot #. */
#define nbic_regs(slotid)  (caddr_t)(0xf0ffffe8 | ((slotid)<<25)

mydriver_var_t my_var[SLOTCOUNT];
. . .
    sp = &my_var[slotid];
    sp->nbic_addr=(unsigned char *)map_addr(nbic_regs(slotid), 24);
. . .

int my_intr(void)
{
    int  slotid;
    volatile unsigned char *intr_reg, *mask_reg;
    /*
     * Figure out if we really handle this interrupt.
     * Interrupts at level 5 are polled, so we have to
     * check the interrupt byte associated with the
     * hardware slots we control.  slotid 1 corresponds
     * to hardware slot 2, slotid 2 to hardware slot 4,
     * and slotid 3 to hardware slot 6.
     */
    for (slotid = 1; slotid < SLOTCOUNT; slotid++)
    {
        if (my_var[slotid].is_ours) {
            intr_reg = my_var[slotid].nbic_addr;
            if (*intr_reg & SLOT_INTR_BIT)
                break;
        }
    }

    /* If we couldn't handle the interrupt, leave now. */
    if (slotid == SLOTCOUNT)
        return FALSE;        /* Poll code should try next handler */
        /*
         * At this point some device-dependent code is necessary to
         * reset the interrupt condition so that the device does
         * not continue to try to interrupt the CPU.  Here we
         * disable the interrupt by clearing the mask bit.
         */
        mask_reg = intr_reg + 4;
        *mask_reg = 0;  /* Mask it off. */

        /* Handle the interrupt. */
        . . .

        /*  Schedule a routine to react to the interrupt.*/
        kern_serv_callout(&instance,
                          (void (*)(void *))my_func,
                          (void *)sp);

        return TRUE;             /* We fielded the interrupt, so no
                                    other driver should be polled. */
    }

    . . .
        install_polled_intr(I_BUS, my_intr);
```

Considerations for Message-Based Kernel Servers

A message-based loadable kernel server can have one of two interfaces: a server interface or a handler interface. MiG automatically produces server interfaces; you have to do some additional hand coding to turn the MiG-generated server interface into a handler interface. Handler interfaces have a performance advantage because you can allocate only as much space as you need for the reply message.

Server and handler interfaces both take two parameters, the first of which is an incoming message. The second argument for a server interface is the outgoing message; for a handler interface, it's an integer or pointer to data.

To convert a server to a handler, you need to make three files:

- A header file to define the handler routine and the data type that helps do server-to-handler translation
- A handler code file that mimics (and includes) the *xxx***Server** file produced by MiG
- An implementation file that contains the code that does the real work

For an example of converting servers to handlers, see the files under **/NextLibrary/Documentation/NextDev/Examples/ServerVsHandler**.

Warning: If you're using MiG, don't name any of your files *xxx*.**h**, where the MiG subsystem name is *xxx*. (MiG will overwrite it.)

A message-based server that receives out-of-line data can't directly access the data. The data is inaccessible because it appears in the server's task's address map, but your server uses the kernel's address map instead of its own task's map. To read or write out-of-line data, you can call **vm_write()** to copy all or part of the data into the kernel map.

Considerations for UNIX-Based Kernel Servers

To use UNIX-based servers, you must provide the proper entry points and insert your 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.*

UNIX Entry Points

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

Character Device Entry Points

```
/* from <sys/conf.h> */
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 routine that handles an **open()** system call.
d_close	A pointer to the server routine that handles a **close()** system call.
d_read	A pointer to the server routine that handles a **read()** system call.
d_write	A pointer to the server routine that handles a **write()** system call.
d_ioctl	A pointer to the server routine 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 routine that handles a **select()** system call. If your device is ready for reading or writing, this routine 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 routine that handles memory mapping of device space to user space. This routine must return the page number of the passed offset. It's typically found in frame buffers.
d_getc	Not supported for user-written servers; used for console devices.
d_putc	Not supported for user-written servers; used for console devices.

Block Device Entry Points

```
/* from <sys/conf.h> */
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 routine that handles an **open**() system call.
d_close	A pointer to the server routine that handles a **close**() system call.
d_strategy	A pointer to the server routine that eventually handles **read**() and **write**() system calls.
d_dump	A pointer to the server routine 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 routine 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 routine called by **kern_loader**.

Below is an example of 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 must 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;
```

Functions Supplied by NeXT

Your loadable kernel server can't use user-level functions; every function it uses must be defined in the kernel. Included in the kernel are all the functions and macros listed in Chapter 5, "C Functions," and almost all Mach Kernel Functions, which are listed in the *Operating System Software* manual. (The only Mach Kernel Function that doesn't work is **mach_error**(), since 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, since they're not 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 when you use routines that reference threads and virtual memory maps.

Communicating with the Hardware

This section discusses how to read and write hardware registers, especially those required on NeXTbus boards. Remember, you should test every NeXTbus hardware access with the NeXTbus Probe before you put the code to do it in your server.

NeXTbus boards can transfer data in up to three sizes: bytes (8 bits), halfwords (16 bits), and words (32 bits). You usually store bytes in **unsigned char** structures, halfwords in **unsigned short** structures, and words in **unsigned int** structures.

Warning: 68040-based systems don't allow you to read or write using a larger data structure than the device width. For example, you can't write a 32-bit quantity into an 8-bit register; you must use an 8-bit quantity such as an **unsigned char**.

Burst transfers (transfers of 16 bytes with one instruction) are turned off by default for addresses that are mapped using **map_addr()**. (This is a side effect of **map_addr()** making the memory non-cacheable, as is desirable when dealing with devices.) To perform a burst read or write on an address mapped with **map_addr()**, you must write assembly code that calls the 68040's MOVE16 instruction. Because MOVE16 reads or writes the bytes in 32-bit chunks, your hardware must be 32 bits wide. Note that the 68030 doesn't have a MOVE16 instruction and doesn't support burst writes. You can also enable burst transfers while using the NeXTbus Probe by turning "Cache Inhibit" off.

For more information on the NeXTbus, see the *NeXTbus Specificatio*n. For information on the NeXTbus Interface Chip (NBIC), see the *NeXTbus Interface Chip Specification.*

Mapping from Physical to Virtual Memory Addresses

Use the routine **map_addr()** to turn a physical address into a virtual memory address that your server can use. For example, to set a flag on a 32-bit register at address 0x100 in a NeXTbus board's slot address space (physical address 0xf*s*000100), you would use the following code:

```
#define REG_ADDR 0xf0000100
volatile unsigned int *reg;

reg = (unsigned int *)map_addr(REG_ADDR | (slot_number << 24), 4);
*reg |= FLAG;
```

For a description of the physical address space, see the "NeXTbus Address Space" section in Chapter 1, "Overview."

NeXTbus Byte Ordering

Depending on your hardware, bytes might be swapped within a word. For example, if you read a word on your NeXTbus board that contains "0x12345678", your server might see "0x78563412".

You can determine whether byte order is an issue for your server by consulting the designers of and documentation for your board, and by testing reads and writes using the NeXTbus Probe. The NeXTbus Probe does not change byte ordering, so if the reads and writes work with the NeXTbus Probe, they should work the same way in your server.

The code below is an example of reversing byte order. In this example, we want to check whether bit 7 in a word is 1 by ORing the word with 0x00000080. However, on a board that swaps bytes, we have to switch the first and fourth bytes (0x80 and 0x00), producing 0x80000000.

```
volatile unsigned int *my_register;
int reg_size = 4;   /* # of bytes in the space to be mapped */
my_register = (unsigned int *)map_addr((MY_ADDR | (slot_num << 24)),
    reg_size);

if (probe_rb(my_register))
    /* Make sure the Valid field is set */
    if ( !(*my_register & 0x80000000)) /* byte-reversed 00000080 */
        return FALSE;
```

CPU Board NBIC Registers

There's only one NBIC register on the CPU board that you might need to write to. This register, the NBIC Control register, has three defined bits. The only bit you might need to modify, however, is the Store Forward bit.

NBIC Control Register

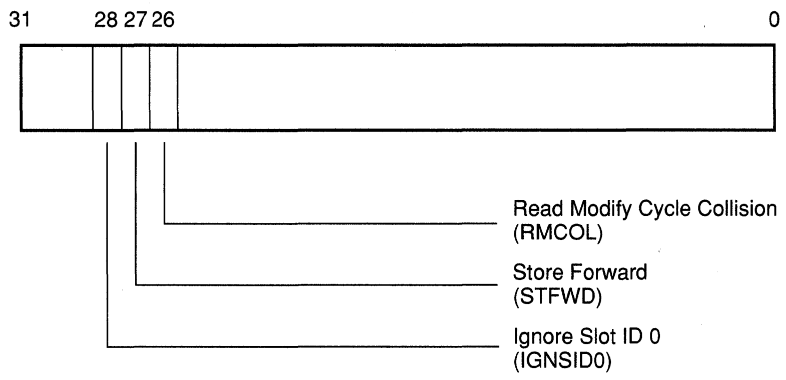
31
28 27 26
0
Read Modify Cycle Collision (RMCOL)
Store Forward (STFWD)
Ignore Slot ID 0 (IGNSID0)

Figure 2-1. NBIC Control Register

The CPU board's NBIC Control register is at address 0x02020000.

Bit 28, Ignore Slot ID 0 (IGNSID0), controls how much NeXTbus address space a board uses. It's set to 1 during initialization so that the CPU board takes up two slots worth of addresses.

Bit 27 is the Store Forward (STFWD) control bit. At power up, it's enabled (set to 1). When enabled, Store Forward causes the CPU board's NBIC to immediately acknowledge writes, without waiting for the other NeXTbus board to write the data. This is called a store and forward write transaction. It speeds up transaction time, since the CPU doesn't have to wait for your board to write data. The disadvantage of store and forward write is that you won't receive any notification of write errors. Thus, unless your hardware is completely reliable, store and forward write can be dangerous.

Bit 26 is the Read Modify Cycle Collision (RMCOL) bit. It's not appropriate to access this register on the CPU board.

NeXTbus Board Registers

On every NeXTbus board, six bytes of identification and interrupt information are at addresses 0xf*s*ffffe8 to 0xf*s*fffffc, where *s* is the board's slot number.

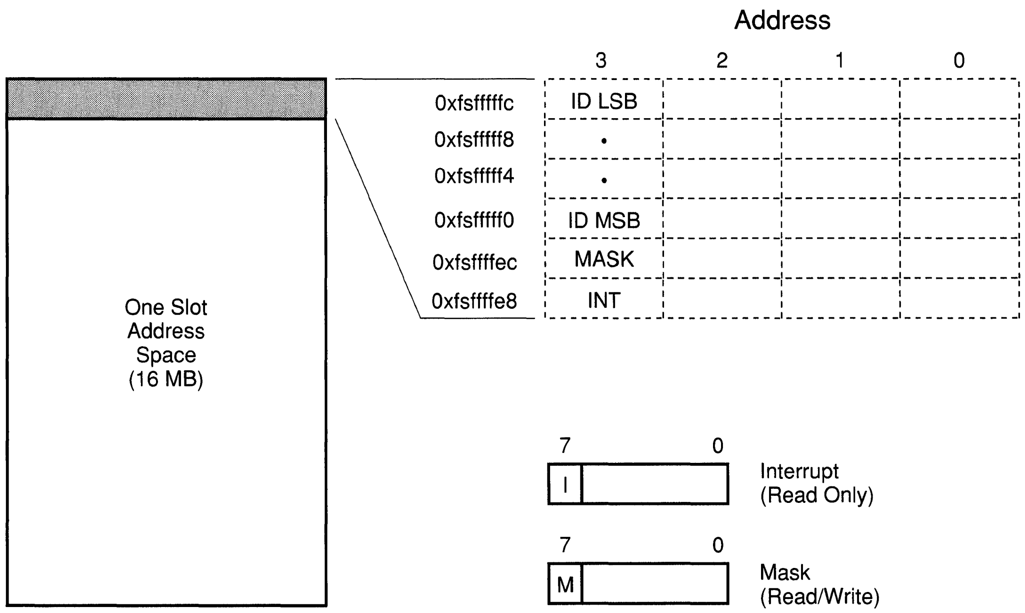

Figure 2-2. Sample NeXTbus Slot Address Space

If your NeXTbus board uses an NBIC, then addresses 0xf*s*fffff0 through 0xf*s*fffffc correspond to the NBIC ID register, address 0xf*s*ffffec is the Interrupt register, and 0xf*s*ffffe8 is the Interrupt Mask register.

Identification Bytes (NBIC ID Register)

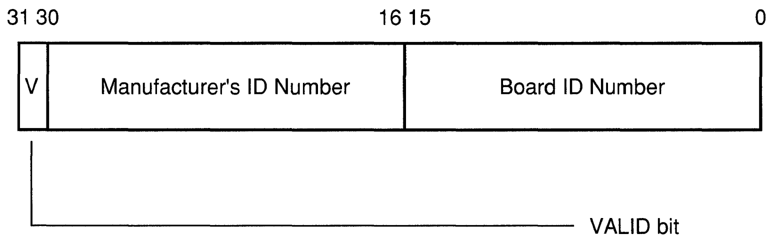

Figure 2-3. NBIC ID Register

These bytes give information that identifies the type of NeXTbus board. Your server should read them during its initialization to see whether it should take control of this board.

The four Identification bytes are read-only bytes that contain a board ID number, a manufacturer's ID number, and a VALID bit. Unless the VALID bit is 1, none of the identification and interrupt information is valid. When the VALID bit is 1, the ID numbers are valid and your server can use them to identify the board.

Since each byte is mapped to a separate 32-bit word, your server has to read four separate words (or bytes, if your hardware supports byte-wide reads) to get all the identification information. Figure 2-4 shows the Identification byte locations in the NeXTbus slot address space.

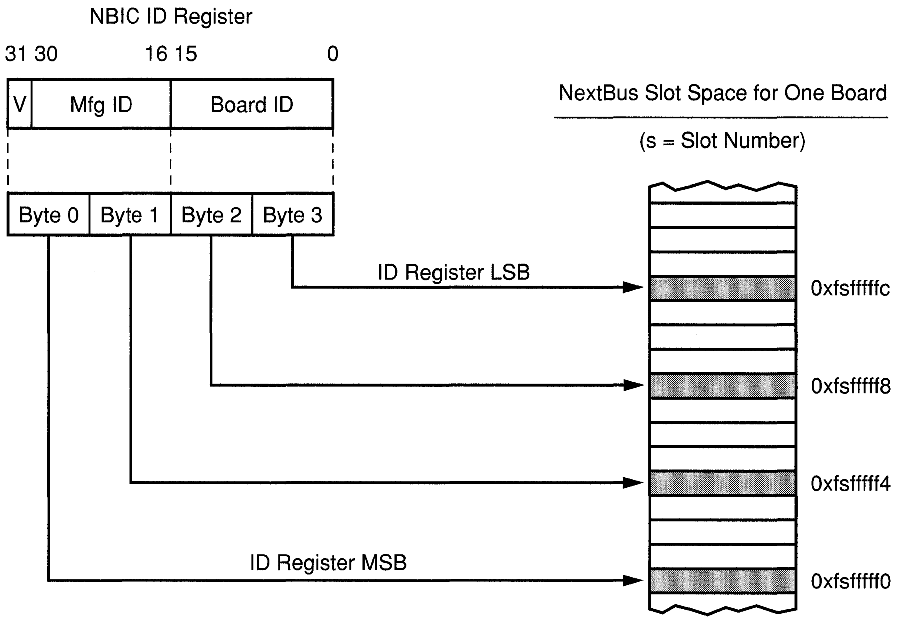

Figure 2-4. Identification Bytes in the NeXTbus Slot Space

If your server can't perform byte reads, you have to find the correct byte in each word or halfword. If your server doesn't have to reverse the byte order, you can simply address the last byte of the word as an **unsigned char**. If you do have to reverse the byte order, you must right-shift each word or halfword to move the byte you want to read from the most significant to the least significant byte.

The following example shows how to read the Identification bytes.

```
volatile unsigned int *id_start;
int id_size = 16;   /* # of bytes in the space to be mapped */

id_start=(unsigned int *)map_addr((0xf0fffff0 | (slotid << 25)),
    id_size);

if (probe_rb(id_start))
{
    if (ids_ok(id_start, MY_MFG_ID, MY_BRD_ID))
        my_var[slotid].present = TRUE;
}

#define HI_BYTE(n) ((n) >> 8)
#define LO_BYTE(n) ((n) & 0xff)
```

```
/*
 * Check the ID on the remote board to see if it matches ours.  The
 * ID fields are spread out over four words from 0xFsFFFFF0 through
 * 0xFsFFFFFC, with the valid data appearing in bits 0 through 7.
 */
boolean_t
ids_ok(volatile unsigned int *id_begin, int board_id, int mfg_id)
{
    volatile unsigned int *current_word = id_begin;

    /* Make sure the Valid field is set */
    if ( !(*current_word & 0x00000080))
        return FALSE;

    /* test high byte of mfg ID, ignoring the Valid field */
    if ( ((*current_word) & 0x7f) != HI_BYTE(mfg_id) )
        return FALSE;

    ++current_word;  /* Move to next word */
    /* test low byte of the mfg ID */
    if ( (*current_word) != LO_BYTE(mfg_id) )
        return FALSE;

    ++current_word;  /* Move to next word */
    /* test high byte of the board ID */
    if ( (*current_word) != HI_BYTE(board_id) )
        return FALSE;

    ++current_word;  /* Move to next word */
    /* test low byte of the board ID */
    if ( (*current_word) != LO_BYTE(board_id) )
        return FALSE;

    return TRUE;
}
```

Interrupt Byte

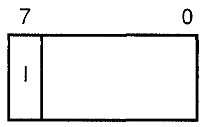

Figure 2-5. NBIC Interrupt Register

This byte contains a bit that shows whether the board wants to interrupt the CPU. Your server's interrupt handler can read this bit to determine whether it needs to handle an interrupt.

The Interrupt byte is a read-only byte at address 0xf*s*ffffe8, where *s* is the slot ID. This byte has only one significant bit, bit 7. Bit 7 is 1 when the board wants to interrupt, and 0 when it doesn't.

Note: The value of bit 7 doesn't depend on whether interrupts are enabled or disabled. Even if interrupts are disabled, and thus this board can't interrupt the CPU, bit 7 will be 1 if the board wants to interrupt.

Interrupt Mask Byte

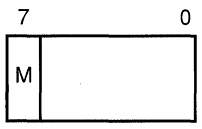

Figure 2-6. NBIC Interrupt Mask Register

Use this byte to enable and disable interrupts.

The Interrupt Mask byte is a read/write byte at address 0xf*s*ffffec, where *s* is the slot ID. As in the Interrupt byte, bit 7 is the only meaningful bit. When bit 7 is 1 (the default), the board can interrupt. If your server writes a 0 to bit 7, the board stops interrupting and can't interrupt again until the server writes a 1 to it.

After your server handles an interrupt, you might want to disable interrupts for a while so that the hardware won't keep interrupting the CPU.

Workaround for Intercepted NeXTbus Addresses

If your server needs to access the addresses that are intercepted by the CPU board (0x*s*200c000 to 0x*s*200cfff, where *s* is 2, 4, or 6), the NBIC on your board must be configured to accept the addresses for the next higher slot (3, 5, or 7).

You can't directly configure the NBIC from your server; it must be done locally on your board. Specifically, the board must set the IGNSID0 bit (bit 28) of its own NBIC Control register to 1. Then, in your server, you can address both the slot and board address spaces of the board with the next higher slot number.

Chapter 3
Testing and Debugging Kernel Servers

As for any driver, testing and debugging are the hardest parts of writing a kernel server. Most of the tools described in this chapter are mentioned in other places, but they're gathered together here to review the methods you can use.

Booting the Computer

When you boot the computer your server is running on, use the **-p** option. This option makes the Panic window stay up when the kernel panics, forcing the system not to reboot until you want it to. How to use the Panic window is described later in this chapter.

Generating Interrupts

At times, your system might freeze due to bugs in your server or mistakes while running KGDB. If this happens and you don't see a Panic window, first try to generate a non-maskable interrupt (NMI), as described below. If this doesn't work, then as a last resort you can reset the CPU.

Generating a Non-Maskable Interrupt (NMI)

To generate an NMI, use Command-Command-~. (Hold down both Command keys and press the key at the upper left of the numeric keypad.) The NMI mini-monitor window appears.

Resetting the CPU

Warning: Use this only as a last resort! Resetting the CPU can damage the file system.

To reset the CPU, hold down the Command and Alternate keys at the lower left of the keyboard, and press the * key on the upper right of the numeric keypad. This causes the machine to reboot immediately. Rebooting will take longer than usual because the file system will be checked.

The System Console

You can view the output of **printf**() statements in your server by keeping the system console window open. To open a console window, choose the Console command from the Workspace Manager™'s Tools menu.

Another way to use the console is to log in with the user name **console** (without a password). This will make the whole screen act like a UNIX terminal that receives all console messages. After you log in, you can enter commands at the shell prompt.

Logging in as **console** is useful when the only thing you want to run is your server, so you don't need the overhead of a windowing environment. This is often true for the slave computer when you're debugging your server with KGDB.

The NMI Mini-Monitor Window

The NMI mini-monitor window is useful for looking at the output of kernel **printf**() calls and for getting a prompt on the master when you're running KGDB. See Appendix D, "The Kernel Debugger," for instructions on getting a prompt on the master.

To view the output of kernel **printf**() calls, use the **msg** command. You can limit the number of messages you see by putting =*n* after the command, where *n* is the number of messages you want to see. For example:

```
nmi> msg=3
```

For more information on the NMI mini-monitor window, see the *Network and System Administration* manual.

The Panic Window

The Panic window is similar to the NMI mini-monitor window, but it comes up only as the result of a kernel panic. In this window you can use the **gdb** and **msg** commands, just as in the NMI mini-monitor window.

Because the system brings up this window instead of crashing completely, you can run KGDB on the panicked system, even if the hardware wasn't set up before the panic. All you have to do is connect the master system and the panicked system (either through the network or with a serial cable), start KGDB as usual on the master system (including entering the **kattach** command), and if necessary enter **gdb** at the panicked system's prompt to get a KGDB prompt on the master. From KGDB, you can do a backtrace to see what routine caused the panic.

After you're done using KGDB, enter **continue** at the Panic window. This will give the system a chance to shut down cleanly if it can.

The ROM Monitor Window

The ROM monitor lets you examine and change the contents of hardware addresses on the CPU board. You should use the NeXTbus Probe to look at addresses that aren't on the CPU board.

To be able to access hardware addresses, you must get into the ROM monitor just after the system is powered on, before it has booted. Otherwise, you'll access virtual addresses, not physical addresses. To enter the ROM monitor, do the following:

1. If your computer is on, press the Power key to turn it off.

Locate the ~ key on the numeric keypad. (You'll need to press it along with a Command key soon after you turn the power back on.)

2. Turn the computer back on by pressing the Power key.

3. *Within three seconds* of the appearance of the message "Loading from disk" or "Loading from network," hold down the Command key and press the ~ key on the numeric keypad.

You should now be in the ROM monitor. You can use the **e** command to look at addresses on the CPU board.

For more information on using the ROM monitor, see Appendix E, "The ROM Monitor and NMI Mini-Monitor."

The NeXTbus Probe

With the NeXTbus Probe (an application in the NeXTbus Development Kit), you can read or write registers while you're testing or debugging a NeXTbus driver. You can use the repeat and auto-increment features to clear areas of memory on the board or for other repetitive tasks.

The NeXTbus Probe lets you choose the board or slot address space that you want to look at. For this reason, you should specify an *offset* into this address space, not the whole address.

Other Tools

KGDB, the kernel debugger, is an important debugging tool. You can use it not only during the normal debugging cycle, but also later on to examine kernel panics caused by your server. Appendix D gives detailed instructions on how to start and use KGDB.

kl_log is useful for getting log messages from your server. If you wish, you can instead write a program that calls **kern_loader_log_level**() and **kern_loader_get_log**(). Appendix C, "The Kernel-Server Log Command," describes how to use **kl_log.**

vm_stat(1) is useful for seeing how many pages are wired down. It can help you find out whether your server is growing too large.

Chapter 4
Network-Related Kernel Servers

Network-related kernel servers, also known as *network modules*, need special functions and interfaces in addition to the ones available to all loadable kernel servers. This chapter discusses how to write network modules. The special functions that network modules can use are described in detail in Chapter 5, "C Functions," under the section "Network Functions."

The NeXT Mach kernel contains support for linking in the following types of network modules:

- *Network device drivers.* A network device driver sends and receives packets to and from some network media.
- *Protocol handlers.* On input, a protocol handler receives packets from network device drivers and forwards the data to the interested programs. On output, the protocol handler takes data from programs, puts the data into packets, and sends these packets to the appropriate network device driver.
- *Packet sniffers.* A packet sniffer merely examines input packets for diagnostic purposes.

If you're familiar with UNIX 4.3BSD networking primitives, you'll find many similarities to what's described in this chapter. The biggest difference is that a common programming interface like the socket mechanism isn't defined. While sockets work well for TCP/IP, they don't generalize well to other protocols.

If you're writing a protocol handler and want to open it up to programmers, you must define your own interface for communication between user programs and your protocol handler.

This chapter first gives an overview of NeXT networking support, and then discusses the objects you'll use in your network module. The next section has details on the routines that you should implement. The chapter ends with notes on implementing specific interfaces.

Overview

Here's a simplified view of what happens when a network packet is received by a NeXT computer:

1. The packet is received by the appropriate network device driver, which puts the packet into a data structure called a *netbuf* (netbufs are discussed below).

2. The driver calls the dispatcher (by calling **if_handle_input()**).

3. The dispatcher polls all registered packet sniffers and protocol handlers until it finds a protocol handler that accepts the packet.

4. If the protocol handler is an IP (Internet Protocol) handler, it sends the packet up to the kernel by calling **inet_queue()**.

When a packet is sent out onto the network, the following events happen:

1. The protocol handler's output function is called. One of the arguments is a netbuf containing the packet to be sent. (This netbuf must have been previously allocated by the network device driver; how netbufs are allocated is described later in this chapter.)

2. The protocol handler calls the appropriate network device driver's output function, passing it the netbuf containing the packet.

3. The device driver puts the packet out onto the network.

Note that there's one extra step in the case of the input packet: A dispatcher is called. This happens because a network device driver doesn't know what its associated protocol handler is, but the protocol handler knows which driver to call. The dispatcher doesn't query the modules in any particular order, except that it queries all packet sniffers before querying any protocol handlers.

Network Objects

The NeXT kernel has two abstractions especially for network modules:

- Network buffers, known as *netbufs*
- Network interfaces, known as *netifs*

Each of these abstractions is discussed below. The associated C functions are described in detail in Chapter 5.

Network Buffers (netbufs)

The NeXT kernel uses netbufs for dealing with network packet buffers. Netbufs are an interface to an abstract sequence of bytes that can be read and written. The sequence has an original starting point and ending point, but these can be changed. An input network packet typically has its starting point advanced as the various headers are pulled off. Similarly, an output packet has its starting point retreated as headers are inserted.

Operating beyond the range of the original starting and ending points isn't currently caught as an error. This means that an outgoing netbuf should be copied into a larger netbuf if the information being added to its top requires more bytes than are available between the current ending point and the original starting point.

Network Interfaces (netifs)

Netifs are used to handle the installation and usage of network modules. Remember that a network module is one of three things: a network device driver, a protocol handler, or a packet sniffer.

Each network module initializes and installs its netif (thus registering itself) by calling **if_attach**(). A network device driver should immediately register itself by calling **if_attach**() at load time. Protocol handlers and packet sniffers, on the other hand, don't have to register themselves until their services are required. They determine whether to register themselves in a callback function that they supply as an argument to the function **if_registervirtual**(). This callback function is called once for each network device driver; it should call **if_attach()** if the module isn't already registered and it wants to receive input packets from the specified driver.

Functions You Should Implement

Besides a callback function, your network module needs to supply certain functions so that other modules can call it. When your network module calls **if_attach**(), you must specify the locations of five functions:

- An initialization function. This should do any initialization that's required to change the module's state to "on."
- An input function. This function should receive packets from lower layers and either consume them or pass them on to other modules.
- An output function. This should send packets from higher layers.
- A getbuf function. This function should provide netbufs for higher layers to use in impending sends.
- A control function. Use this function to provide any necessary operations the above functions don't do.

Note: You should specify null to **if_attach**() for any unimplemented function.

These five functions, along with the callback function, are described in more detail below.

Callback Function

typedef void (*if_attach_func_t)(void *private*, netif_t *realif*)

A callback function is required in protocol handlers and packet sniffers, but isn't appropriate in network device drivers. Its purpose is to determine whether it's interested in the device driver and, if necessary, to register its module (using **if_attach()**). The callback function is called once for each current and future network device driver, so it can keep information about more than one network device driver.

The callback function is specified in the network module's call to **if_registervirtual()**. The *private* argument is the data that was specified in the call to **if_registervirtual()**. *realif* is a pointer to the network device driver for which this function is being called. The following code is an example of a typical callback function for a protocol handler.

```
static void myhandler_attach(void *private, netif_t rifp)
{
    netif_t ifp;
    const char *name;
    int unit;
    void *ifprivate;

    if (strcmp(if_type(rifp), IFTYPE_ETHERNET) != 0) {
        return;
    }

    ifprivate = (void *)kalloc(sizeof(myhandler_private_t));
    name = MYNAME;
    unit = MYUNIT;
    ifp = if_attach(NULL, myhandler_input, myhandler_output,
        myhandler_getbuf, myhandler_control, name, unit, IFTYPE_IP,
        MYMTU, IFF_BROADCAST, NETIFCLASS_VIRTUAL, ifprivate);

    (myhandler_private_t *)if_private(ifp)->rifp = rifp;

    if_control(rifp, IFCONTROL_GETADDR, MYHANDLER_ADDRP(ifp));

    if (verbose) {
        printf("IP protocol enabled for interface %s%d, type
            \"%s\"\n", name, unit, MYDRIVER_TYPE);
    }
    return;
}

void myhandler_config(void)
{
    if_registervirtual(myhandler_attach, NULL);
}
```

Initialization Function

typedef int (*if_init_func_t)(netif_t *netif*);

An initialization function is not required but is often found in network device drivers. It takes a pointer to its module's netif structure and performs any necessary initialization. For example, a network device driver should perform any steps necessary to have its hardware ready to run. You can determine what the integer return value (if any) should be.

```
int mydriver_init(netif_t netif)
{
    unsigned unit = if_unit(netif);
    register struct mydriver_data_t *is = &mydriver_data[unit];

    if (is->is_flags & HW_RUNNING)
        return;

    is->is_flags |= HW_RUNNING;
    /* Initialize software structures and the hardware. */
    /* ... */
    return;
}
```

Input Function

typedef int (*if_input_func_t)(netif_t *netif*, netif_t *realnetif*, netbuf_t *packet*, void **extra*);

An input function is required in protocol handlers and packet sniffers, but not in network device drivers. It takes a pointer to its module's netif (*netif*), a pointer to the calling network device driver (*realnetif*), the input packet, and optional extra data. This function should examine the input packet and decide if it wants the packet. If so, this function should return zero and take responsibility for freeing the packet. Otherwise, this function should return EAFNOSUPPORT to allow other modules to receive the packet. Packet sniffers should always return EAFNOSUPPORT.

For example, an IP handler getting packets from an Ethernet device would check if an Ethernet packet's protocol number is the value for IP. If so, the IP handler should handle the packet and return zero.

Since this function might be called at interrupt priority, it should only queue packets. Another thread should pull the packets off of the queue and process them.

The following code is a typical input function of a protocol handler.

```
static int venip_input(netif_t ifp, netif_t rifp, netbuf_t nb,
    void *extra)
{
    short etype;
    short offset;
    short size;
    trailer_data_t trailer_data;

    /* Do we want packets from this driver? */
    if ((myhandler_private_t *)if_private(ifp)->rifp != rifp) {
        return (EAFNOSUPPORT);
    }

    /*
     * Check fields in the packet to see whether they match
     * the protocol we understand.
     */
    nb_read(nb, MYTYPEOFFSET, sizeof(etype), &etype);
    etype = htons(etype);
    /*
     * Handle ethernet trailer protocol.
     */
    if (etype >= ETHERTYPE_TRAIL &&
        etype < ETHERTYPE_TRAIL + ETHERTYPE_NTRAILER) {
        offset = (etype - ETHERTYPE_TRAIL) * 512;
        if (offset == 0 || (ETHERHDRSIZE + offset +
                    sizeof(trailer_data) >=
                    nb_size(nb))) {
            return (EAFNOSUPPORT);
        }
        nb_read(nb, ETHERHDRSIZE + offset, sizeof(trailer_data),
            &trailer_data);
        etype = htons(trailer_data.etype);
        if (etype != ETHERTYPE_IP &&
            etype != ETHERTYPE_ARP) {
            return (EAFNOSUPPORT);
        }
        size = htons(trailer_data.length);
        if (ETHERHDRSIZE + offset + size > nb_size(nb)) {
            return (EAFNOSUPPORT);
        }
        /*
         * trailer_fix() is a private function that converts trailer
         * packet to regular ethernet packet.
         */
        trailer_fix(nb, offset, size - sizeof(trailer_data));
    }
    switch (etype) {
    case ETHERTYPE_IP:
        nb_shrink_top(nb, ETHERHDRSIZE);
        if_ipackets_set(ifp, if_ipackets(ifp) + 1);
        inet_queue(ifp, nb);
        break;
        /* Put other cases here as necessary. */
        default:
        /*
         * Do not free buf:  let others handle it
         */
        return (EAFNOSUPPORT);
    }
    return (0);
}
```

Output Function

typedef int (*if_output_func_t)(netif_t *netif*, netbuf_t *packet*, void **address*);

All network modules except packet sniffers must have an output function. This function takes a pointer to the module's netif, a pointer to a packet, and an address. How this function works depends on whether it's part of a protocol handler or of a network device driver.

If this function is part of a protocol handler, it should assume the packet and address are strictly protocol-level entities, containing no device-dependent information. The function should add network device information to the packet and call the network device driver's output routine. The netbuf that holds the packet should have been returned by this module's getbuf function, as described below.

If this function is part of a network device driver, it should assume the packet and address are device-level entities. The function should simply deliver the packet to the given device-level address. Its return value should be zero if no error occurred; otherwise, return an error number from the header file **sys/errno.h**.

```
static int venip_output(netif_t ifp, netbuf_t nb, void *addr)
{
    struct sockaddr *dst = (struct sockaddr *)addr;
    struct ether_header eh;
    struct in_addr idst;
    int off;
    int usetrailers;
    netif_t rifp = VENIP_RIF(ifp);
    int error;
```

```
        switch (dst->sa_family) {
        case AF_UNSPEC:
            bcopy(dst->sa_data, &eh, sizeof(eh));
            break;
        case AF_INET:
            idst = ((struct sockaddr_in *)dst)->sin_addr;
            /* ... */
            /*
             * Resolve the en address using arp.  Return 0 if the address
             * wasn't resolved.
             */
             /*
             * XXX:  trailers not supported for output
             */
            eh.ether_type = htons(ETHERTYPE_IP);
            break;
        default:
            nb_free(nb);
            return (EAFNOSUPPORT);
        }
        nb_grow_top(nb, ETHERHDRSIZE);
        nb_write(nb, ETYPEOFFSET, sizeof(eh.ether_type),
              (void *)&eh.ether_type);
        error = if_output(rifp, nb, (void *)&eh.ether_dhost);
        if (error == 0) {
            if_opackets_set(ifp, if_opackets(ifp) + 1);
        } else {
            if_oerrors_set(ifp, if_oerrors(ifp) + 1);
        }
        return (error);
    }
```

Getbuf Function

typedef netbuf_t (*if_getbuf_func_t)(netif_t *netif*);

A getbuf function is required in all modules except packet sniffers. This function returns a netbuf to be used for an impending output call. Only network device drivers should allocate these netbufs. Protocol handlers should instead call the appropriate network device driver's getbuf function to do the allocation. After allocation from the network device driver and before returning the result, the protocol handler should leave enough room at the top of the netbuf for its own output function to later insert a header.

A getbuf function doesn't always have to return a buffer. For example, you might want to limit the number of buffers your module can allocate (say, 200KB worth) so that it won't use up too much wired-down kernel memory. When a getbuf function fails to return a buffer, it should return null.

In a protocol handler:

```
static netbuf_t venip_getbuf(netif_t ifp)
{
    netif_t  rifp = VENIP_RIF(ifp);
    netbuf_t nb;

    nb = if_getbuf(rifp);
    if (nb == NULL) {
        return(NULL);
    }
    nb_shrink_top(nb, ETHERHDRSIZE);
    return(nb);
}
```

In a driver:

```
static netbuf_t engetbuf(struct ifnet *ifp)
{
    if (numbufs == MAXALLOC)
        return(NULL);
    else {
        numbufs++;
        return(nb_alloc(HDR_SIZE + ETHERMTU));
    }
}
```

Control Function

typedef int (*if_control_func_t)(netif_t *netif*, const char **command*, void **data*)

The control function isn't required, but it's useful in all three kinds of network modules. This function performs arbitrary operations, with the character string *command* used to select between these operations. There are five standard operations that you can choose to implement, although you can also define your own. The command strings corresponding to the standard operations are listed in the table below; constants for the strings (such as IFCONTROL_SET_FLAGS for "setflags") are declared in the header file **net/netif.h**.

Command	**Operation**
"setflags"	Request to have interface flags turned on or off.
"setaddr"	Set the address on the interface.
"getaddr"	Get the address of the interface.
"autoaddr"	Automatically set the address of the interface.
"unix-ioctl"	Perform a UNIX **ioctl()** command. This is only for compatibility; **ioctl()** isn't a recommended interface for network drivers. The argument is of type **if_ioctl_t ***, where the **if_ioctl_t** structure contains the UNIX ioctl request (for example, SIOCSIFADDR) in the **ioctl_command** field and the ioctl data in the **ioctl_data** field.

```
static int
venip_control(netif_t ifp, const char *command, void *data)
{
    netif_t rifp = VENIP_RIF(ifp);
    unsigned ioctl_command;
    void *ioctl_data;
    int s;
    struct sockaddr_in *sin = (struct sockaddr_in *)data;

    if (strcmp(command, IFCONTROL_AUTOADDR) == 0) {
        /*
         * Automatically set the address
         */
        if (sin->sin_family != AF_INET) {
            return (EAFNOSUPPORT);
        }
        /* ... */
    } else if (strcmp(command, IFCONTROL_SETADDR) == 0) {
        /*
         * Manually set address
         */
        if (sin->sin_family != AF_INET) {
            return (EAFNOSUPPORT);
        }
        if_flags_set(ifp, if_flags(ifp) | IFF_UP);
        if_init(rifp);
        VENIP_PRIVATE(ifp)->vp_ipaddr = sin->sin_addr;
        /* ... */
    } else {
        /*
         * Let lower layer handle
         */
        return (if_control(rifp, command, data));
    }
    return (0);
}
```

Notes for Specific Interfaces

Ethernet Interfaces

Network device drivers that implement the 10-megabit-per-second Ethernet protocol should register their type as IFTYPE_ETHERNET (defined in the header file **net/etherdefs.h**). One 10Mb Ethernet network device driver comes standard with the NeXT operating system. The type of the address passed to the Ethernet driver's output function for output should be a six-byte character array (which you cast to **void** *).

TCP/IP Interfaces

IP protocol handlers can hand over their input packets to the kernel for processing by calling **inet_queue**().

IP protocol handlers should specify their type as "Internet Protocol" when they call **if_attach**(). The NeXT operating system comes with two TCP/IP modules–one for delivery over Ethernet, and one for delivery over loopback. The type of address used by IP protocol handlers should be **struct sockaddr_in**, which is defined in the header file **netinet/in.h**.

Chapter 5
C Functions

This chapter gives detailed descriptions of the C functions provided by the NeXT Mach operating system for loadable kernel servers. Also included here are some macros that behave like functions. For this chapter, the functions and macros are divided into two groups: general functions and network functions.

Network functions are those that are specifically for network-related kernel servers. All the other functions are under the "General Functions" section.

Within each section, functions are subgrouped with other functions that perform related tasks. These subgroups are described in alphabetical order by the name of the first function listed in the subgroup. Functions within subgroups are also listed alphabetically, with a pointer to the subgroup's description.

For convenience, these functions are summarized in Appendix F, "Summary of Kernel Functions." The summary lists functions by the same subgroups used in this chapter and combines several related subgroups under a heading such as "Time Functions" or "Memory Functions." For each function, the appendix shows the calling sequence.

General Functions

ASSERT()

SUMMARY Panic if an assumption isn't true

SYNOPSIS

void **ASSERT**(int *expression*)

ARGUMENTS

expression: A C expression that's 0 when the assumption isn't true.

DESCRIPTION

ASSERT() is a macro that works only if you specify the DEBUG C preprocessor macro when you compile your server. If *expression* is 0, **ASSERT()** calls **panic()** after printing the line and file that the assertion failed in.

You might want to redefine **panic()** so that **ASSERT()** calls **kern_serv_panic()** when possible. For example:

```
#define panic(s) (curipl() == 0 ? \
    kern_serv_panic((kern_serv_bootstrap_port(&instance), s) \
    : printf("Can't panic: %s\n", s))
```

EXAMPLE

In your makefile:

```
CFLAGS  = ... -DDEBUG
```

In your server:

```
ASSERT(ptr != NULL);
```

SEE ALSO

panic(), **kern_serv_panic()**

assert_wait()

SUMMARY Arrange for a thread to sleep on an event

SYNOPSIS

void **assert_wait**(int *event*, boolean_t *interruptible*)

ARGUMENTS

event: An integer that identifies the event. Typically, this is the address of a structure.

interruptible: Used by **clear_wait()**. If *interruptible* is false and the *interrupt_only* argument to a later call to **clear_wait()** is true, then this thread won't be waked up by that call to **clear_wait()**.

DESCRIPTION

Use this routine before calling **thread_block()**. This routine sets up the event that the thread wants to wait for, but the thread doesn't start sleeping until it executes **thread_block()**.

EXAMPLE

```
extern  hz;

assert_wait(0, FALSE);
thread_set_timeout(hz*2);
thread_block();
```

SEE ALSO

clear_wait(), **thread_block()**, **biowait()**

bcopy()

SUMMARY Copy data into a buffer

SYNOPSIS

void **bcopy**(void **from*, void **to*, int *length*)

ARGUMENTS

from: Start of buffer to be copied from.

to: Start of buffer to be copied to.

length: Number of bytes to copy.

DESCRIPTION

Like the C library **bcopy()** routine, this routine copies bytes from one buffer to another buffer in the same virtual space. **bcopy()** cannot be used to copy data between user space and kernel space. The caller of this routine must have already checked the access rights to this memory and wired it down.

Important: Use **bytecopy()** instead of **bcopy()** if you're copying to or from hardware device space that's only 8 or 16 bits wide. (**bcopy()** often uses 32-bit accesses for efficiency, but the 68040 processor doesn't allow 32-bit accesses to 8- or 16-bit hardware.)

SEE ALSO

bytecopy(), **strcpy()**, **copyin()**, **copyout()**

biodone()

SUMMARY Wake up the routine doing a **biowait()** on a buffer

SYNOPSIS

#import <sys/buf.h>

void **biodone**(struct buf **bp*)

ARGUMENTS

bp: The address of a **buf** structure.

DESCRIPTION

This routine marks the buffer as done and wakes up any threads waiting for it. If the B_DONE flag is already set, **biodone()** panics. Otherwise, if B_CALL is set, **biodone()** clears it and calls the routine pointed to by **bp*–>**b_iodone**. Next, if B_ASYNC is set, **biodone()** releases the buffer pointed to by *bp*; if B_ASYNC isn't set, **biodone()** clears the B_WANTED flag and wakes up all threads that had called **biowait()** on *bp*.

EXAMPLE

```
one_thread(void)
{
    struct buf  mybuf;
    . . .
    biowait (&mybuf);
    . . .
}

other_thread(struct buf *bp)
{
    . . .
    biodone(bp)
   . . .
}
```

SEE ALSO

biowait()

biowait()

SUMMARY Wait until a routine calls **biodone()** on a buffer

SYNOPSIS

#import <sys/buf.h>

void **biowait**(struct buf **bp*)

ARGUMENTS

bp: The address of a **buf** structure.

DESCRIPTION

If the B_DONE flag in the buffer pointed to by *bp* is already set, this routine won't sleep. Otherwise, this routine sleeps until another thread calls **biodone()** on *bp*.

EXAMPLE

```
one_thread(void)
{
    struct buf  mybuf;
    . . .
    biowait (&mybuf);
    . . .
}

other_thread(struct buf *bp)
{
    . . .
    biodone(bp)
   . . .
}
```

SEE ALSO

biodone(), **assert_wait()**

bytecopy()

SUMMARY Copy bytes into a buffer

SYNOPSIS

void **bytecopy**(void **from*, void **to*, int *length*)

ARGUMENTS

from: Start of buffer to be copied from.

to: Start of buffer to be copied to.

length: Number of bytes to copy.

DESCRIPTION

This function is like **bcopy()**, except that it uses only 8-bit instructions to copy data. **bytecopy()**, like **bcopy()** and the C library **bcopy()** routine, copies bytes from one buffer to another buffer in the same virtual space. **bytecopy()** cannot be used to copy data between user space and kernel space. The caller of this routine must have already checked the access rights to this memory and wired it down.

Note: This function is less efficient than **bcopy()**, so you should use **bcopy()** unless you're copying to or from hardware device space that's only 8 or 16 bits wide.

SEE ALSO

bcopy(), **strcpy()**, **copyin()**, **copyout()**

bzero()

SUMMARY Zero out a region of memory

SYNOPSIS

void **bzero**(void **address*, int *length*)

ARGUMENTS

address: The address of the first byte of the region of memory.

length: The number of bytes to write zeros to.

DESCRIPTION

This acts the same as the **bzero**() C library function.

SEE ALSO

bzero() UNIX manual page

clear_wait()

SUMMARY Stop a thread from waiting for an event

SYNOPSIS

#import <sys/sched_prim.h>

void **clear_wait**(thread_t *thread*, int *result*, boolean_t *interrupt_only*)

ARGUMENTS

thread: The thread to wake up.

result: The wakeup result the thread should see.

interrupt_only: If true, don't wake up the thread unless **assert_wait**() was called with *interruptible* set to true.

DESCRIPTION

Use this routine to wake up a thread that's waiting for an event (as the result of **assert_wait**() and **thread_block**()), whether or not the event has happened. If *interrupt_only* is false or if **assert_wait**() was called with *interruptible* set to false, then the thread is guaranteed to wake up. The thread will receive *result* when it calls **thread_wait_result**().

EXAMPLE

```
void        new_thread(void);
extern      hz;
char        data;
thread_t    thread1;
. . .
{
    . . .
    thread1 = (thread_t)current_thread();
    kernel_thread(current_task(), new_thread);
    assert_wait(&data, FALSE);
    thread_block();
    printf("Wait result:  %d\n",
        thread_wait_result());
}

void new_thread()
{
    . . .
    clear_wait(thread1, THREAD_AWAKENED, FALSE);
    . . .
} /* new_thread */
```

SEE ALSO

assert_wait(), **thread_block()**, **thread_wait_result()**, **thread_wakeup()**, **us_untimeout()**

copyin()

SUMMARY Copy bytes from user to kernel space

SYNOPSIS

int **copyin**(void **from*, void **to*, int *length*)

ARGUMENTS

from: The start of the region in user space.

to: The start of the region in kernel space.

length: The number of bytes to copy from user to kernel space.

DESCRIPTION

Returns 0 if successful, –1 otherwise.

SEE ALSO

bcopy(), **copyout()**

copyout()

SUMMARY Copy bytes from kernel to user space

SYNOPSIS

int **copyout**(void **from*, void **to*, int *length*)

ARGUMENTS

from: The start of the region in kernel space.

to: The start of the region in user space.

length: The number of bytes to copy from kernel to user space.

DESCRIPTION

The same as **copyin()**, except the direction of the copy is reversed.

SEE ALSO

bcopy(), **copyin()**

curipl()

SUMMARY Get the current interrupt level

SYNOPSIS

int **curipl**()

DESCRIPTION

This function returns the CPU interrupt level, which is a number between 0 and 7.

EXAMPLE

```
#define panic(s) (curipl() == 0 ? \
    kern_serv_panic((kern_serv_bootstrap_port(&instance), s) \
    : printf("Can't panic: %s\n", s))
```

SEE ALSO

spl*n*(), **splx()**

current_task()

SUMMARY Get the current task

SYNOPSIS

task_t **current_task**()

DESCRIPTION

This macro returns the task structure for the current task. Use **current_task()** whenever you need to refer to the task in which your kernel server executes. Don't use this to refer to memory unless you specifically want the task's native memory map, and not the kernel map that your server uses.

EXAMPLE

```
kernel_thread(current_task(), new_thread);
```

SEE ALSO

kernel_thread()

DELAY()

SUMMARY Busy-wait for a certain number of microseconds

SYNOPSIS

#import <next/machparam.h>

void **DELAY**(unsigned int *usecs*)

ARGUMENTS

usecs: The number of microseconds to delay for.

DESCRIPTION

This macro makes the processor loop for the number of microseconds specified in the argument. Interrupts are not disabled by this routine, so surround **DELAY()** with **spl*n*()** and **splx()** if interrupts need to be disabled. Because the microsecond resolution clock is used to count the spin interval, the delay is independent of CPU instruction clock speed.

This macro doesn't sleep, so it's safe to use in interrupt handlers. It's often used to wait for the hardware.

EXAMPLE

```
/* set the hardware register for at least 100 microseconds */
hardware_register = 1;
DELAY(100);
hardware_register = 0;
```

SEE ALSO

us_timeout(), **us_abstimeout**(), **us_untimeout**(), **microtime**(), **microboot**(), **spl*n***(), **splx**()

install_polled_intr()

SUMMARY Install an interrupt handler for a polled device

SYNOPSIS

#import <next/cpu.h>
#import <next/autoconf.h>

int **install_polled_intr**(int *which*, int (**my_intr*)())

ARGUMENTS

which: Specifies the device and interrupt level. For devices attached through the NeXTbus interface, this should be the constant I_BUS.

my_intr: The routine in your server that handles this interrupt.

DESCRIPTION

This function installs an interrupt handler; you can later remove this interrupt handler by calling **uninstall_polled_intr**().

This routine returns 0 if the call is successful, or –1 if the interrupt level specified by *which* isn't capable of interrupt polling.

EXAMPLE

```
device_interrupt() {
    if (interrupt_is_for_us) {
        /* -process interrupt- */
        return (1);  /* say interrupt was for us */
    }
    else
        return (0);  /* it must be for someone else */
}
```

```
device_initialize() {
    install_polled_intr(I_BUS, device_interrupt);
    . . .
}
```

SEE ALSO

uninstall_polled_intr()

kalloc()

SUMMARY Allocate wired-down kernel memory

SYNOPSIS

void ***kalloc**(int *size*)

ARGUMENTS

size: The size in bytes to be allocated.

DESCRIPTION

This routine is guaranteed to return wired-down memory of the requested size. You can't call **kalloc**() from an interrupt handler because it might sleep.

Memory returned isn't guaranteed to be aligned in any way unless size is a multiple of the page size (in which case the memory is page-aligned). If you need to ensure alignment, you should allocate twice what you need and align the address you start with to the boundary you want. Memory isn't guaranteed to be contained on the same physical page unless you allocate in multiples of the page size and keep track of the page location of addresses you use. The page size is dynamic, and there's currently no way to get its value from inside the kernel; however, on 680x0-based machines, 8192 is guaranteed to be an integer multiple of the page size in bytes.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kalloc(sizeof (my_data_t));
. . .
kfree(arg, sizeof (my_data_t));
```

SEE ALSO

kfree(), **kget**()

kernel_thread()

SUMMARY Start a new kernel thread in the specified task

SYNOPSIS

thread_t **kernel_thread**(task_t *task*, void (**start*)())

ARGUMENTS

task: For loadable kernel servers, this must be **current_task()**.

start: The first routine to be called by the new thread.

DESCRIPTION

This routine can sleep, so don't call it from an interrupt handler. The new thread uses the kernel's address map, but the loadable kernel server's task.

EXAMPLE

```
void new_thread(void);

kernel_thread(current_task(), new_thread);

void new_thread()
{
    /* Do something, then (if necessary) shut down */
    thread_terminate((thread_t)thread_self());
    thread_halt_self();
} /* new_thread */
```

SEE ALSO

current_task()

kern_serv_bootstrap_port()

SUMMARY Get the port used to initialize your server

SYNOPSIS

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

port_t **kern_serv_bootstrap_port**(kern_server_t **ksp*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION

This routine returns the port that the kernel uses to initialize (or "bootstrap") your server when it's loading it. Normally, the only reason to use this port is as an argument to **kern_serv_panic**().

EXAMPLE

```
bootstrap_port=kern_serv_bootstrap_port(&instance);
kern_serv_panic(bootstrap_port, "Couldn't send message");
```

SEE ALSO

kern_serv_panic(), kern_serv_local_port(), kern_serv_notify_port(), kern_serv_port_set()

kern_serv_callout()

SUMMARY Run a function in the server's main thread

SYNOPSIS

kern_return_t **kern_serv_callout**(kern_server_t **ksp*, void (**func*)(void *), void **arg*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

func: The kernel server function to be called.

arg: The argument to be passed to *func*.

DESCRIPTION

This function provides a way for interrupt handlers to call functions in the same kernel server that may sleep or deal with a user context. The function *func* is called with argument *arg* at some point in the future.

EXAMPLE

```
void mydriver_func(mydriver_data_t data)
{
    . . .
}

kern_serv_callout ((kern_server_t *)&instance, mydriver_func,
                   (void *)arg);
```

RETURN

KERN_SUCCESS: The callout was scheduled successfully.

KERN_RESOURCE_SHORTAGE: The callout couldn't be scheduled.

kern_serv_local_port()

SUMMARY Determine which port the kernel just received a message on

SYNOPSIS

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

port_t **kern_serv_local_port**(kern_server_t **ksp*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION

This function returns the port on which the kernel just received a message in your server's behalf. The only time this function is useful is when your server was just loaded as the result of a message to one of its ports.

EXAMPLE

```
port=kern_serv_local_port(&instance);
if (port==debug_port)
    debug=TRUE;
```

SEE ALSO

kern_serv_notify_port(), kern_serv_port_set()

kern_serv_log()

SUMMARY Put a message in the kernel server's error log

SYNOPSIS

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

void **kern_serv_log**(kern_server_t **ksp*, int *log_level*, char **format*, *arg1*, ..., *arg5*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

log_level: A number indicating the urgency of this log entry. Higher numbers indicate greater urgency, but the particular range of numbers used in a kernel server is up to the writer of that kernel server.

format: A string containing formatting information. See **printf**().

arg1, ..., *arg5*: Arguments to be printed. (If you don't specify all five arguments, the compiler will display a warning, but the call will still succeed.) See **printf**().

DESCRIPTION

This function puts a message in the error log. The message can be retrieved by a user process that calls **kern_loader_get_log**() or by the command **kl_log**.

EXAMPLE

```
kern_serv_log(&instance, 5, "Reset value of timeout to %d\n", time,
    0, 0, 0, 0);
```

SEE ALSO

log(), **printf**()

kern_serv_notify()

SUMMARY Ask to receive notification messages about a certain port

SYNOPSIS

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

kern_return_t **kern_serv_notify**(kern_server_t **ksp*, port_t *reply_port*, port_t *request_port*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

reply_port: The port that should receive the notification messages. This should normally be the value returned by **kern_serv_notify_port()**.

request_port: The port we want to be notified about.

DESCRIPTION

This function requests that notification messages about *request_port* be sent to *reply_port*. The types of notification messages are defined in the header file **sys/notify.h**.

EXAMPLE

```
notify_port=kern_serv_notify_port(&instance);
kern_serv_notify(&instance, notify_port, bootstrap_port);
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_FAILURE: The same *reply_port*, *request_port* pair has already been entered.

SEE ALSO

kern_serv_notify_port()

kern_serv_notify_port()

SUMMARY Get the notification port of this server

SYNOPSIS

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

port_t **kern_serv_notify_port**(kern_server_t **ksp*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION

This routine returns this server's notification port, which can be used in calls to **kern_serv_notify()**.

EXAMPLE

```
notify_port=kern_serv_notify_port(&instance);
kern_serv_notify(&instance, notify_port, bootstrap_port);
```

SEE ALSO

kern_serv_notify()

kern_serv_panic()

SUMMARY Unload this server without panicking the system

SYNOPSIS

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

kern_return_t **kern_serv_panic**(port_t **bootstrap_port*, panic_msg_t *message*)

ARGUMENTS

bootstrap_port: This server's bootstrap port, which is returned by **kern_serv_bootstrap_port()**.

message: A string to be added to the panic message that's logged.

DESCRIPTION

This routine unloads the server after logging a message in the kernel server loader's log. The message is logged at the priority LOG_WARNING and contains the name of the server that called this routine, followed by *message*.

EXAMPLE

```
kern_serv_panic(bootstrap_port,
    "my_server_main: received bad return from msg_receive");
```

RETURN

KERN_SUCCESS: The server will be unloaded.

SEE ALSO

ASSERT(), **panic()**, **kern_serv_bootstrap_port()**

kern_serv_port_gone()

SUMMARY Notify the kernel that a port will be deleted

SYNOPSIS

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

void **kern_serv_port_gone**(kern_server_t **ksp*, port_name_t *port*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that will be deleted.

DESCRIPTION

Use this function to make sure that the kernel won't send any more messages to a certain port.

EXAMPLE

```
/*
 * Deallocate transmit port.
 */
kern_serv_port_gone(&instance, my_dev->xmit_port);
(void)port_deallocate((task_t)task_self(), my_dev->xmit_port);
my_dev->xmit_port = PORT_NULL;
```

SEE ALSO

kern_serv_port_proc(), **kern_serv_port_serv()**

kern_serv_port_proc()

SUMMARY Set which function is a port's handler

SYNOPSIS

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

kern_return_t **kern_serv_port_proc**(kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that the function should be associated with.

function: The function that handles messages sent to *port*.

arg: An integer to be passed in the call to *function* whenever *port* receives a message.

DESCRIPTION

Use this function to register a message-receiving function in a handler-style (not server-style) loadable kernel server. This function provides the functionality of the HMAP load command to your server.

EXAMPLE

```
/* Create the port. */
r = port_allocate((task_t)task_self(), &port_name);
if (r != KERN_SUCCESS)
    kern_serv_panic(&instance, "couldn't allocate a port");
else printf("Created port %d\n", port_name);

/* Specify which function is its handler. */
r = kern_serv_port_proc(&instance, port_name,
    (port_map_proc_t)myhandler, 0);
if (r != KERN_SUCCESS) {
    kern_serv_panic("port_allocate failed (%d)\n", r);
    exit(1);
}

/* . . . */

kern_serv_port_gone(&instance, port_name);
port_deallocate((task_t)task_self(), port_name);
port_name = PORT_NULL;
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_RESOURCE_SHORTAGE: No more port to function mappings are available for your loadable kernel server.

KERN_NOT_RECEIVER: You don't have receive rights for *port*.

KERN_INVALID_ARGUMENT: *port* isn't a valid port.

SEE ALSO

kern_serv_port_gone(), kern_serv_port_serv()

kern_serv_port_serv()

SUMMARY Set which function is a port's message server

SYNOPSIS

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

kern_return_t **kern_serv_port_serv**(kern_server_t **ksp*, port_all_t *port*,
    port_map_proc_t *function*, int *arg*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

port: The port that the function should be associated with.

function: The function that handles messages sent to *port*.

arg: An integer to be passed in the call to *function* whenever *port* receives a message.

DESCRIPTION

This function is just like **kern_serv_port_proc()** except that it registers a function with a server-style, as opposed to a handler-style, interface. This routine performs the same function as the SMAP load command.

EXAMPLE

```
/* Create the port. */
r = port_allocate((task_t)task_self(), &port_name);
if (r != KERN_SUCCESS)
    kern_serv_panic(&instance, "couldn't allocate a port");
else printf("Created port %d\n", port_name);

/* Specify which function is its server. */
r = kern_serv_port_serv(&instance, port_name,
    (port_map_proc_t)myserv, 0);
if (r != KERN_SUCCESS) {
    kern_serv_panic("port_allocate failed (%d)\n", r);
    exit(1);
}

/* . . . */

kern_serv_port_gone(&instance, port_name);
port_deallocate((task_t)task_self(), port_name);
port_name = PORT_NULL;
```

RETURN

KERN_SUCCESS: The call succeeded.

KERN_RESOURCE_SHORTAGE: No more port to function mappings are available for your loadable kernel server.

KERN_NOT_RECEIVER: You don't have receive rights for *port*.

KERN_INVALID_ARGUMENT: *port* isn't a valid port.

SEE ALSO

kern_serv_port_gone(), **kern_serv_port_proc()**

kern_serv_port_set()

SUMMARY Get the port set

SYNOPSIS

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

port_set_name_t **kern_serv_port_set**(kern_server_t **ksp*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

DESCRIPTION

This function returns the name of the port set on which messages to the kernel server arrive. The kernel listens to this port set on behalf of your kernel server. Usually, this function is used after you've temporarily removed a port from the port set, and you need the name of the port set as a parameter to **port_set_add()** so you can put the port back into the port set.

EXAMPLE

```
/* Don't accept any more requests until we get rid of the old ones. */
port_set_remove((task_t)task_self(), dev->xmit_port);

. . . /* Get rid of some old requests. */

/* Re-enable listening on the port. */
port_set_add((task_t)task_self, kern_serv_port_set(&instance),
    dev->xmit_port);
```

SEE ALSO

kern_serv_port_gone(), **kern_serv_port_proc()**, **kern_serv_port_serv()**

kern_serv_unwire_range()

SUMMARY Unwire the specified range of memory in the kernel map

SYNOPSIS

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

kern_return_t **kern_serv_unwire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

address: A virtual address in the kernel map.

size: The size in bytes to be wired down.

DESCRIPTION

This routine makes a region of kernel memory subject to swapping. Usually, you'd call it when you're preparing to deallocate the memory with **vm_deallocate**().

RETURN

KERN_SUCCESS: The call succeeded.

KERN_INVALID_ARGUMENT: The range of memory wasn't wired down.

SEE ALSO

kalloc(), **kfree**(), **kget**(), **kern_serv_wire_range**()

kern_serv_wire_range()

SUMMARY Wire down the specified range of memory in the kernel map

SYNOPSIS

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

kern_return_t **kern_serv_wire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)

ARGUMENTS

ksp: The address of the first field (which must be of type **kern_server_t**) in the server's instance variable.

address: A virtual address in the kernel map.

size: The size in bytes to be wired down.

DESCRIPTION

This routine wires down a range of kernel memory. Usually you'd call it after you've copied out-of-line data into the kernel map.

RETURN

KERN_SUCCESS: The call succeeded.

SEE ALSO

kalloc(), **kget()**, **kern_serv_unwire_range()**

kfree()

SUMMARY Free memory that was allocated using **kalloc()** or **kget()**

SYNOPSIS

void **kfree**(void **address*, int *size*)

ARGUMENTS

address: The memory to be freed.

size: The size in bytes to be freed.

DESCRIPTION

The memory freed will be available for subsequent **kalloc()** and **kget()** calls only if the size they specify is the same as *size*.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kalloc(sizeof (my_data_t));
. . .
kfree(arg, sizeof (my_data_t));
```

SEE ALSO

kalloc(), **kget()**

kget()

SUMMARY Try to quickly allocate wired-down kernel memory

SYNOPSIS

void ***kget**(int *size*)

ARGUMENTS

size: The size in bytes to be allocated. This size, rounded up to the nearest power of 2, must be less than the page size (default 8192 bytes) or the kernel will panic.

DESCRIPTION

Use this routine in interrupt handlers to try to get kernel memory. If no memory of the appropriate size can be allocated without blocking, **kget()** returns 0. Otherwise, it returns the address of the chunk of memory.

EXAMPLE

```
my_data_t *arg;

arg = (my_data_t *)kget(sizeof (my_data_t));
if (arg != 0)
{ . . .
    kfree(arg, sizeof (my_data_t));
}
```

SEE ALSO

kalloc(), **kfree()**

lock_alloc(), lock_free()

SUMMARY Create or destroy a lock

SYNOPSIS

lock_t **lock_alloc()**
void **lock_free**(lock_t *lock*)

ARGUMENTS

lock: The lock to be freed.

DESCRIPTION

lock_alloc() returns a pointer to a new lock. Before you use the lock, you should initialize it by calling **lock_init()**.

lock_free() frees the lock structure pointed to by *lock*.

See **lock_done**() for information on using locks.

SEE ALSO

lock_done(), **lock_init**(), **lock_read**(), **lock_write**(), **simple_lock_alloc**(), **simple_lock_free**()

lock_done()

SUMMARY Release a read or write lock

SYNOPSIS

void **lock_done**(lock_t *lock*)

ARGUMENTS

lock: A pointer to the lock that the reader or writer wants to release.

DESCRIPTION

The **lock**_*xxx*() routines provide reader/writer synchronization. Any number of readers can read, as long as no one has a lock for writing. A writer can get a lock only if no there are no existing reader or writer locks. Once a writer tries to get a lock, no more readers can get the lock, and the writer gets the lock as soon as the last reader releases its lock. The writer sleeps or busy-waits until it can get a lock; you determine which it does when you initialize the lock.

Use the **lock_done**() routine to relinquish a read or write lock.

EXAMPLE

```
lock_write(lock1);
/* write to the protected data */
lock_done(lock1);
```

SEE ALSO

lock_alloc(), **lock_free**(), **lock_init**(), **lock_read**(), **lock_write**(), **simple_lock_unlock**()

lock_free() → See lock_alloc()

lock_init()

SUMMARY Initialize a lock

SYNOPSIS

void **lock_init**(lock_t *lock*, boolean_t *can_sleep*)

ARGUMENTS

lock: A pointer to the lock that the reader or writer wants to initialize.

can_sleep: If true, threads waiting to acquire a lock can sleep. If false, threads will busy-wait while trying to acquire a lock. This should be usually be true.

DESCRIPTION

Use this routine to initialize a lock when you first create it. See **lock_done()** for a description of how locking works. Use **lock_alloc()** to create the lock.

EXAMPLE

```
lock_t lock1 = lock_alloc();
lock_init(lock1, TRUE);

/*. . .*/

lock_free(lock1);
```

SEE ALSO

lock_alloc(), **lock_done()**, **lock_free()**, **lock_read()**, **lock_write()**, **simple_lock_init()**

lock_read()

SUMMARY Get a lock for reading

SYNOPSIS

void **lock_read**(lock_t *lock*)

ARGUMENTS

lock: A pointer to the lock that the reader wants to get.

DESCRIPTION

Use this routine to get a lock for reading some data. If a writer holds or is waiting for a lock, you won't get the lock until the writer is done. Otherwise, you'll get the lock, even if other readers have it locked.

EXAMPLE

```
lock_t lock1 = lock_alloc();

lock_init(lock1, TRUE);
.
.
.
lock_read(lock1);
if (DONE_READING)
    lock_done(lock1);
```

SEE ALSO

lock_alloc(), **lock_done()**, **lock_free()**, **lock_init()**, **lock_write()**, **simple_lock**

lock_write()

SUMMARY Get a lock for writing

SYNOPSIS

void **lock_write**(lock_t *lock*)

ARGUMENTS

lock: A pointer to the lock that the writer wants to get.

DESCRIPTION

Use this routine to get a lock for writing some data. If another writer has or is waiting for a lock, you won't get the lock until the writer is done. If any readers have locks, you won't get the lock until every reader releases its lock.

EXAMPLE

```
lock_t lock1;

lock1 = lock_alloc();
lock_init(lock1, TRUE);
.
.
.
lock_write(lock1);
if (DONE_WRITING)
    lock_done(lock1);
```

SEE ALSO

lock_alloc(), **lock_done()**, **lock_free()**, **lock_init()**, **lock_read()**, **simple_lock**

log()

SUMMARY Write a message in the system log buffer

SYNOPSIS

#import <sys/syslog.h>

int **log**(int *level*, char **format*, *arg*, ...)

ARGUMENTS

level: The priority of the information. These priorities are defined in the header file **sys/syslog.h**.

format: A string containing formatting information. See **printf**().

arg, ...: Arguments to be printed. See **printf**().

DESCRIPTION

Prints the time of day and who sent the message (for loadable kernel servers, it's usually sent by Mach). This routine doesn't sleep, so it can be called by interrupt routines. If no process is currently reading the system log, **log()** also writes to the console. This function always returns zero.

EXAMPLE

```
log(LOG_INFO, "My driver: device %s attached\n", device_type);
```

SEE ALSO

kern_serv_log(); **printf**(); UNIX manual pages for **syslog**, **openlog**, **closelog**, **setlogmask**, and **syslogd**

map_addr()

SUMMARY Convert a physical address to a virtual address

SYNOPSIS

caddr_t **map_addr**(caddr_t *address*, int *size*)

ARGUMENTS

address: The physical address.

size: The number of bytes to map.

DESCRIPTION

This function returns a virtual address that corresponds to *address*. At least *size* bytes of hardware addresses are mapped into virtual memory. (Currently, **map_addr**() maps in multiples of the page size, nominally 8192 bytes.)

If you aren't sure whether a hardware address is implemented, you should use **map_addr**() to get a virtual address for it, and then call **probe_rb**() on the virtual address.

EXAMPLE

```
volatile unsigned int *my_reg;

my_reg = (unsigned int *)map_addr(REG_ADDRESS, 4);
if (probe_rb (my_reg))
    *my_reg |= A_FLAG;
else
    printf("Hardware at physical address 0x%x caused bus error\n",
        REG_ADDRESS);
```

SEE ALSO

probe_rb()

microboot()

SUMMARY Return the amount of time since the system booted

SYNOPSIS

#import <sys/time.h>

void **microboot**(struct timeval **tvp*)

ARGUMENTS

tvp: Pointer to a **timeval** structure.

DESCRIPTION

This routine returns the best possible estimate of the time since system boot, to microsecond resolution.

EXAMPLE

```
/* Detect when system has been booted for over one hour. */
struct timeval tv;

microboot(&tv);
if (tv.tv_sec > 1*60*60)
    printf("System has been booted for over an hour\n");
else
    printf("System has not been booted for an hour yet\n");
```

SEE ALSO

us_timeout(), us_abstimeout(), us_untimeout(), microtime(), DELAY()

microtime()

SUMMARY Return the current time

SYNOPSIS

#import <sys/time.h>

void **microtime**(struct timeval **tvp*)

ARGUMENTS

tvp: Pointer to a **timeval** structure.

DESCRIPTION

This routine returns in **tvp* the best possible estimate of the current time, to microsecond resolution. The time returned is the same as what **gettimeofday**() returns: the number of seconds and microseconds since January 1, 1970.

EXAMPLE

```
struct timeval  tv;

microtime(&tv);
printf ("current time: %d secs, %d usecs\n",
    tv.tv_sec, tv.tv_usec);
```

SEE ALSO

us_timeout(), us_abstimeout(), us_untimeout(), microboot(), DELAY(), gettimeofday(2)

panic()

SUMMARY Hang the system and bring up the Panic window

SYNOPSIS

void **panic**(char **string*)

ARGUMENTS

string: The message to be printed to the console, message log, and **/usr/adm/messages**.

DESCRIPTION

Calling **panic**() brings up the Panic window (similar to the NMI mini-monitor window) and either hangs or reboots the system, depending on whether you booted with the **-p** option. See Chapter 3 for information on the Panic window.

Instead of using **panic**() you should use **kern_serv_panic**() when possible, since it doesn't cause the whole system to panic. However, **kern_serv_panic**() can't be called when the interrupt level is greater than 0.

EXAMPLE

```
if (curipl() == 0)
    kern_serv_panic(bootstrap_port, "Couldn't get resource");
else
    panic("mydriver:  Couldn't get resource");
```

SEE ALSO

ASSERT(), **kern_serv_panic**()

printf()

SUMMARY Display a message on the console

SYNOPSIS

int **printf**(char **format* [, *arg1*, ...])

ARGUMENTS

format: The format string. It's just like the C library **printf**() routine's format string, except that only **%s**, **%c**, **%x** (==**%X**), **%d** (==**%D**==**%u**), and **%o** (==**%O**) are recognized.

arg1, ...: Arguments to be formatted according to the *format* string.

DESCRIPTION

This routine is a scaled-down version of the C library **printf()** routine. Output goes not only to the console, but also to the message buffer and to **/usr/adm/messages**. Since **printf()** disables interrupts while printing messages, all system activities are suspended while it writes to the console.

Although **printf()** is safe to call in interrupt handlers, its output isn't guaranteed to print on the console. The message buffer, however, should be up-to-date. You can read the message buffer using the **msg** command in the NMI mini-monitor.

printf() always returns zero.

SEE ALSO

sprintf(), **kern_serv_log()**, **log()**

probe_rb()

SUMMARY Check whether an address exists

SYNOPSIS

int **probe_rb**(void **address*)

ARGUMENTS

address: A virtual address that refers to a physical address.

DESCRIPTION

This routine returns 1 if *address* refers to a valid hardware address, 0 otherwise.

EXAMPLE

```
volatile unsigned int *my_reg;

my_reg = (unsigned int *)map_addr(REG_ADDRESS, 4);
if (probe_rb (my_reg))
    *my_reg |= A_FLAG;
else
    printf("Hardware at physical address 0x%x caused bus error\n",
        REG_ADDRESS);
```

SEE ALSO

map_addr()

simple_lock()

SUMMARY Get a simple lock

SYNOPSIS

void **simple_lock(**simple_lock_t *lock***)**

ARGUMENTS

lock: A pointer to the simple lock.

DESCRIPTION

Simple locks are simple spin-loops that implement exclusive locks. They're designed to be used when you're going to hold the lock for only a short time and/or when you can't sleep.

If the someone else already has the lock, this routine will busy-wait until it gets the lock.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);
/* . . . */

/* Set a lock before manipulating a data structure */
simple_lock(slock);
mydriver->data1 = VALUE;
simple_unlock(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO

simple_lock_alloc(), **simple_lock_free()**, **simple_lock_init()**, **simple_unlock()**, **lock_read()**, **lock_write()**

simple_lock_alloc(), simple_lock_free()

SUMMARY Allocate or free a simple lock

SYNOPSIS

simple_lock_t **simple_lock_alloc**()
void **simple_lock_free**(simple_lock_t *lock*)

ARGUMENTS

lock: The simple lock to be freed.

DESCRIPTION

simple_lock_alloc() returns a pointer to a new simple lock. Before you use the simple lock, you should initialize it by calling **simple_lock_init**().

simple_lock_free() frees the structure pointed to by *lock*.

See **simple_lock**() for information on how to use simple locks.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO

simple_lock(), simple_lock_init(), simple_unlock(), lock_alloc(), lock_free()

simple_lock_init()

SUMMARY Initialize a simple lock

SYNOPSIS

void **simple_lock_init**(simple_lock_t *lock*)

ARGUMENTS

lock: A pointer to the simple lock to be initialized.

DESCRIPTION

Use this routine to initialize a new simple lock. You should use **simple_lock_alloc()** to create the lock.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO

simple_lock(), **simple_lock_alloc()**, **simple_lock_free()**, **simple_unlock()**, **lock_init()**

simple_unlock()

SUMMARY Release a simple lock

SYNOPSIS

void **simple_unlock**(simple_lock_t *lock*)

ARGUMENTS

lock: A pointer to the simple lock to be released.

EXAMPLE

```
simple_lock_t   slock;

slock = simple_lock_alloc();
simple_lock_init(slock);
/* . . . */

/* Set a lock before manipulating a data structure */
simple_lock(slock);
mydriver->data1 = VALUE;
simple_unlock(slock);

/* . . . */
simple_lock_free(lock1);
```

SEE ALSO

simple_lock(), **simple_lock_alloc()**, **simple_lock_free()**, **simple_lock_init()**, **lock_done()**

spl*n*()

SUMMARY Set the CPU interrupt level to *n*

SYNOPSIS

int **spl0()**, **spl1()**, **spl2()**, **spl3()**, **spl4()**, **spl5()**, **spl6()**, **spl7()**

DESCRIPTION

The **spl*n*()** macros set the hardware interrupt level of the CPU to level *n*. This means that devices whose hardware interrupt level is greater than *n* will be serviced immediately on an interrupt. Devices with interrupt levels equal to or less than *n* will not be serviced until the CPU interrupt level drops below the device interrupt level.

Because the NMI and power fail interrupts are always serviced, **spl6()** has the same effect as **spl7()** on NeXT computers. **spl0()** sets the CPU interrupt level to the lowest level, enabling all interrupts. The table below shows the interrupts that occur at each hardware interrupt level.

The **spl*n*()** routines return an integer suitable for use with the **splx()** routine to reset the CPU interrupt level.

Interrupt Level	**Interrupts at This Level**
7	NMI (non-maskable interrupt) key sequence Power fail interrupt (non-maskable)
6	System timer timeout interrupt All DMA completion interrupts (except video out)
5	RS-422 (serial) device interrupt NeXTbus interrupts
4	DSP device interrupt
3	Disk device interrupt SCSI device interrupt Laser printer device interrupt Ethernet transmit/receive device interrupts (not DMA) Sound out underrun or sound in overrun Video out DMA completion interrupt Monitor control interrupt Keyboard/mouse event Power on switch Network device interrupts
2	Network-related software interrupts Software interrupt 1
1	Software clock interrupts (timeouts) Software interrupt 0

EXAMPLE

```
#define spl_NB()  spl5() /* NeXTbus interrupt level */
int s;

s = spl_NB();            /* Lock out all NeXTbus interrupts. */
/* Do something that requires that we not be interrupted. */
splx(s);                 /* Return to the previous interrupt level */
```

SEE ALSO

curipl(), splx()

splx()

SUMMARY Reset the CPU interrupt level

SYNOPSIS

splx(int *priority*)

ARGUMENTS

priority: The value returned from the previous call to **spl*n*()**.

DESCRIPTION

This macro returns the hardware priority interrupt level to the level that it was before issuing the last **spl*n*()** command. You must set *priority* to the value returned from the previous call to **spl*n*()**; setting it to anything else doesn't work.

EXAMPLE

```
#define spl_NB()  spl5()  /* NeXTbus interrupt level */
int  s;

s = spl_NB();    /* Lock out all NeXTbus interrupts. */
/* Do something that requires that we not be interrupted. */
splx(s);         /* Return to the previous interrupt level */
```

SEE ALSO

curipl(), **spl*n*()**

sprintf()

SUMMARY Put characters into a string

SYNOPSIS

int **sprintf**(char **string*, char **format* [, *arg1*, ...])

ARGUMENTS

string: The string that you want to put the characters in.

format: The format string. It's just like the C library **printf()** routine's format string, except that only **%s**, **%c**, **%x** (==**%X**), **%d** (==**%D**==**%u**), and **%o** (==**%O**) are recognized.

arg1, ...: Arguments to be formatted according to the *format* string.

DESCRIPTION

This works like the C library routine **sprintf**(), except that it handles only the formats allowed by the kernel **printf**() routine.

SEE ALSO

printf(), **strcat**(), **strcpy**()

strcat()

SUMMARY Concatenate two strings

SYNOPSIS

char ***strcat**(char **string1*, char **string2*)

ARGUMENTS

string1: The string to add the second string to. It must have enough space for *string2* plus a null character.

string2: The string to copy to the end of *string1*.

DESCRIPTION

This acts the same as the **strcat**() C library function. It returns a pointer to *string1*.

SEE ALSO

sprintf(), **strcpy**(), **strlen**()

strcmp()

SUMMARY Compare two strings

SYNOPSIS

int **strcmp**(char **string1*, char **string2*)

ARGUMENTS

string1: The string to be compared to *string2*.

string2: The string being compared against.

DESCRIPTION

This acts the same as the **strcmp**() C library function. It returns an integer greater than, equal to, or less than 0, depending on whether *string1* is lexicographically greater than, equal to, or less than *string2*.

SEE ALSO

strlen()

strcpy()

SUMMARY Copy one string to another

SYNOPSIS

char ***strcpy**(char **to*, char **from*)

ARGUMENTS

to: The string to copy *from* to. It must have enough space to hold all of *from*, including the null character.

from: The string to copy to *to*.

DESCRIPTION

This acts the same as the **strcpy**() C library function. It returns a pointer to *to*.

SEE ALSO

sprintf(), **strcat**(), **strlen**()

strlen()

SUMMARY Get the length of a string

SYNOPSIS

int **strlen**(char **string*)

ARGUMENTS

string: The string you want the length of.

DESCRIPTION

This acts the same as the **strlen**() C library function. It returns the number of non-null characters in *string*.

SEE ALSO

strcmp()

suser()

SUMMARY Check whether the user is the superuser

SYNOPSIS

int **suser()**

DESCRIPTION

This routine is valid only for UNIX-style servers because message-based servers don't have access to user process information. If the user is the superuser, this returns 1 and sets a flag bit indicating that the process has used superuser privileges. Otherwise, it returns 0 and sets **u.u_error** to EPERM.

thread_block()

SUMMARY Put the current thread to sleep.

SYNOPSIS

void **thread_block()**

DESCRIPTION

This routine blocks the current thread from execution. You must call **assert_wait()** before calling **thread_block()**. This thread can be waked up by a timeout (set using **thread_set_timeout()**), by a call to **clear_wait()**, or by a call to **thread_wakeup()**.

EXAMPLE

```
extern  hz;
. . .
splx(s);
assert_wait(0, FALSE);
thread_set_timeout(hz/2);
thread_block();
```

SEE ALSO

assert_wait(), clear_wait(), thread_set_timeout(), thread_wakeup()

thread_halt_self()

SUMMARY Stop the current thread

SYNOPSIS

void **thread_halt_self()**

DESCRIPTION

This makes the current thread stop running. You must first call **thread_terminate()** on the current thread.

EXAMPLE

```
thread_terminate((thread_t)thread_self());
thread_halt_self();
```

SEE ALSO

thread_terminate()

thread_set_timeout()

SUMMARY Set a timer before calling **thread_block()**

SYNOPSIS

void **thread_set_timeout**(int *ticks*)

ARGUMENTS

ticks: The number of ticks to wait for. *n****hz** = *n* seconds' worth of ticks.

DESCRIPTION

This routine sets a timer for the current thread. If you use it, you must call it between **assert_wait()** and **thread_block()**. Use the external variable **hz** (ticks per second) to convert from seconds into ticks. The thread will be waked up in *ticks*/**hz** seconds with a value of THREAD_TIMED_OUT as its wait result (obtained by calling **thread_wait_result()**).

EXAMPLE

```
splx(s);
assert_wait(0, FALSE);
thread_set_timeout(hz*2);
thread_block();
```

SEE ALSO

thread_block(), **thread_wait_result()**, **us_timeout()**, **us_abs_timeout()**

thread_sleep()

SUMMARY Sleep until the specified event occurs

SYNOPSIS

void **thread_sleep(**int *event*, simple_lock_t *lock*, boolean_t *interruptible***)**

ARGUMENTS

event: The event to wait for. This should be a unique integer, such as the address of a buffer.

lock: The simple lock to unlock before calling **thread_block()**.

interruptible: Used by **clear_wait()**. If *interruptible* is false and the *interrupt_only* argument to a later call to **clear_wait()** is true, then this thread won't be waked up by that call to **clear_wait()**.

DESCRIPTION

This is a convenient way to sleep without manually calling **assert_wait()**. This routine causes the current thread to wait until the specified event occurs. The specified lock is unlocked before releasing the CPU.

This routine is equivalent to:

```
assert_wait(event, interruptible);  /* assert event */
simple_unlock(lock);                /* release the lock */
thread_block();                     /* block ourselves */
```

EXAMPLE

```
extern void     thread_wakeup();
struct timeval  tv = {1, 0};

s = splmine();
simple_lock(data.slock);
if (SOME_CONDITION) {
    /* wait */
    us_timeout(thread_wakeup, (int)&data, &tv,
        CALLOUT_PRI_SOFTINT0);
    thread_sleep((int)&data, data.slock, TRUE);
}
simple_unlock(data.slock);
splx(s);
```

SEE ALSO

assert_wait(), **simple_unlock()**, **thread_block()**, **thread_wakeup()**

thread_wait_result()

SUMMARY Get the wait result of the current thread

SYNOPSIS

#import <mach_types.h>
#import <kern/sched_prim.h>

int **thread_wait_result()**

DESCRIPTION

A thread that wakes up for any reason has a result in its thread structure; **thread_wait_result()** returns this result. Possible return values are defined in the header file **kern/sched_prim.h** as THREAD_AWAKENED, THREAD_TIMED_OUT, THREAD_INTERRUPTED, THREAD_SHOULD_TERMINATE, and THREAD_RESTART.

EXAMPLE

```
assert_wait(&data, FALSE);
thread_block();
printf("Wait result:  %d\n",
    thread_wait_result());
```

SEE ALSO

thread_set_timeout(), **thread_wakeup()**

thread_wakeup()

SUMMARY Wake up all threads that are waiting for the specified event

SYNOPSIS

void **thread_wakeup**(int *event*)

ARGUMENTS

event: The event that was specified in the matching **assert_wait**() or **thread_sleep**() call.

DESCRIPTION

The threads that this macro wakes up have THREAD_AWAKENED in their wait result (obtainable by calling **thread_wait_result**()).

Warning: This function must be called at an interrupt level of IPLSCHED or below. You can use **curipl**() to determine what the interrupt level is.

EXAMPLE

```
extern void     thread_wakeup();
struct timeval  tv = {1, 0};      /* 1-second timeout */

s = splmine();
simple_lock(data.slock);
if (SOME_CONDITION) {
    /* wait */
    us_timeout(thread_wakeup, (int)&data, &tv,
        CALLOUT_PRI_SOFTINT0);
    thread_sleep((int)&data, data.slock, TRUE);
}
simple_unlock(data.slock);
splx(s);
```

In an interrupt handler for a NeXTbus driver:

```
if ( (sp->flags & SERVER_THREAD_PAUSED) != 0 )
{
    kern_serv_callout( &instance, thread_wakeup,
        (void *)&sp->server_thread );
}
```

SEE ALSO

assert_wait(), thread_sleep()

uninstall_polled_intr()

SUMMARY Remove an interrupt handler for a polled device

SYNOPSIS

#import <next/cpu.h>
#import <next/autoconf.h>

int **uninstall_polled_intr**(int *which*, int (**my_intr*)())

ARGUMENTS

which: Specifies the device and interrupt level. For devices attached through the NeXTbus interface, this should be the constant I_BUS.

my_intr: The routine in your server that handles this interrupt.

DESCRIPTION

This function removes *my_intr* from the list of functions that are called when an interrupt occurs at interrupt level *which.*

This routine returns 0 if the call is successful. It returns –1 if the interrupt level specified by *which* isn't capable of interrupt polling, or if *my_intr* isn't found.

EXAMPLE

```
device_cleanup()
{
    /* . . . */
    uninstall_polled_intr(I_BUS, device_interrupt);
    /* . . . */
}
```

SEE ALSO

install_polled_intr()

us_abstimeout()

SUMMARY Start a microsecond-accurate timeout in absolute time

SYNOPSIS

#import <sys/time.h>
#import <sys/callout.h>

void **us_abstimeout**(int (**function*)(), vm_address_t *arg*, struct timeval **tvp*, int *priority*)

ARGUMENTS

function: The function to be called.

arg: A single argument that will be passed to *function*.

tvp: Pointer to a **timeval** structure containing the time, in seconds and microseconds relative to system boot time, when *function* is to be called.

priority: Priority that *function* is executed at. This should almost always be the value CALLOUT_PRI_SOFTINT0. Other values may not be supported by future releases.

DESCRIPTION

This routine is used to schedule the execution of a function at a specific time in the future. A single argument may be specified to be passed to the function when it's called. The execution time is specified relative to the system boot time. Although the **timeval** structure allows microsecond resolution to be specified, the time is rounded up to the system clock tick interval so that multiple requests will be batched together (thus reducing overhead).

In any event, it's unrealistic to expect microsecond execution accuracy, because of the interference from interrupt latency. The *priority* argument specifies how the function will be executed. The value CALLOUT_PRI_SOFTINT0 means that the routine will be run from a software interrupt rather than at the interrupt level of the system clock. This prevents the function from delaying interrupts at or below the system clock level. The function will be executed only once per call to **us_abstimeout**().

Use **us_untimeout**() to unschedule the execution of the function before it has been run.

EXAMPLE

```
hour_after_boot (void *arg);
{
    printf("%s", arg);
}

/* schedule execution one hour after boot */
struct timeval tv;

tv.tv_sec = 1*60*60;
tv.tv_usec = 0;
us_abstimeout(hour_after_boot, "arg", &tv, CALLOUT_PRI_SOFTINT0);
```

SEE ALSO

us_timeout(), us_untimeout(), microtime(), microboot(), DELAY()

us_timeout()

SUMMARY Start a microsecond-accurate timeout

SYNOPSIS

#import <sys/time.h>
#import <sys/callout.h>

void **us_timeout**(int (**function*)(), vm_address_t *arg*, struct timeval **tvp*, int *priority*)

ARGUMENTS

function: The function to be called.

arg: A single argument that will be passed to *function*.

tvp: Pointer to a **timeval** structure containing the time, in seconds and microseconds, from the time **us_timeout**() is called to the time *function* is to be called.

priority: Priority that the function is executed at. This should almost always be the value CALLOUT_PRI_SOFTINT0. Other values might not be supported by future releases.

DESCRIPTION

Use this routine to schedule the execution of a function at a specific time in the future. A single argument may be specified to be passed to the function when it's called. The execution time is specified relative to the current time. Although the **timeval** structure allows microsecond resolution to be specified, the time is rounded up to the system clock tick interval so that multiple requests will be batched together (thus reducing overhead).

In any event, it's unrealistic to expect microsecond execution accuracy, because of the interference from interrupt latency. The *priority* argument specifies how the function will be executed. The value CALLOUT_PRI_SOFTINT0 means that the routine will be run from a software interrupt rather than at the interrupt level of the system clock. This prevents the function from delaying interrupts at or below the system clock level. The function will be executed only once per call to **us_timeout**().

Use **us_untimeout**() to unschedule the execution of the function before it has been run.

EXAMPLE

```
void every_second (void *arg);
{
    struct timeval tv;

    printf("arg = %d\n", (int)arg);

    /* Reschedule execution for one second from now. */
    tv.tv_sec = 1;
    tv.tv_usec = 0;
    us_timeout(every_second, (int)arg + 1, &tv, CALLOUT_PRI_SOFTINT0);
}

/* Schedule initial execution in one second. */
struct timeval  tv = {1, 0};

us_timeout(every_second, 0, &tv, CALLOUT_PRI_SOFTINT0);
```

SEE ALSO

us_abstimeout(), **us_untimeout()**, **microtime()**, **microboot()**, **DELAY()**

us_untimeout()

SUMMARY Unschedule a timeout

SYNOPSIS

#import <sys/callout.h>

boolean_t **us_untimeout**(int (**function*)(), vm_address_t *arg*)

ARGUMENTS

function: The function that was to be called.

arg: A single argument that was to be passed to the function.

DESCRIPTION

This routine is used to unschedule a call to a function previously arranged by **us_timeout()** or **us_abstimeout()**. Only one instance of the *function*, *arg* pair is removed, so it may be necessary to call **us_untimeout()** multiple times. The routine has no effect if the *function*, *arg* pair, isn't found or if the function is already being executed. **us_untimeout()** returns true if the timeout was found and unscheduled, otherwise it returns false.

EXAMPLE

```
hour_after_boot (void *arg);
{
    . . .
}

/* schedule execution one hour after boot */
struct timeval tv;

tv.tv_sec = 1*60*60;
tv.tv_usec = 0;
us_abstimeout(hour_after_boot, "arg", &tv, CALLOUT_PRI_SOFTINT0);
. . .
if (WE_CHANGED_OUR_MIND)
    us_untimeout(hour_after_boot, "arg");
```

SEE ALSO

us_timeout(), **us_abstimeout()**, **microtime()**, **microboot()**, **DELAY()**

Network Functions

htonl(), htons(), ntohl(), ntohs()

SUMMARY Convert values between host and network byte order

SYNOPSIS

#include <netinet/in.h>

u_long **htonl**(u_long *hostlong*)
u_short **htons**(u_short *hostshort*)
u_long **ntohl**(u_long *netlong*)
u_short **ntohs**(u_short *netshort*)

DESCRIPTION

These functions and macros simulate the C library functions of the same name. See the UNIX manual page for **byteorder** for more information.

if_attach()

SUMMARY Initialize and install a new netif

SYNOPSIS

#include <net/netif.h>

netif_t **if_attach**(if_init_func_t *init_func*, if_input_func_t *input_func*,
 if_output_func_t *output_func*, if_getbuf_func_t *getbuf_func*,
 if_control_func_t *control_func*, const char **name*, unsigned int *unit*,
 const char **type*, unsigned int *mtu*, unsigned int *flags*, netif_class_t *class*,
 void **private*)

ARGUMENTS

init_func: This module's initialization function.

input_func: This module's input function.

output_func: This module's output function.

getbuf_func: This module's buffer allocation function.

control_func: This module's control function.

name: A constant string that names module (for example,"en").

unit: The unit number of this module (for example, **0**).

type: A constant string that describes the type of this module (for example, "10MB Ethernet").

mtu: The maximum transfer unit (for example, **1500** for Ethernet). This is the maximum amount of data your module can send or receive. Note that protocol-level modules must return the minimum of either the protocol limit or the network device driver's limit (minus header information).

flags: Initial flags for the interface. Possible values are:

IFF_UP:	If true, this interface is working
IFF_BROADCAST:	If true, this interface supports broadcast
IFF_LOOPBACK:	If true, this interface is local only
IFF_POINTTOPOINT:	If true, this is a point-to-point interface

class: The class of this interface. Possible values are:

NETIFCLASS_REAL:	Network driver
NETIFCLASS_VIRTUAL:	Protocol handler
NETIFCLASS_SNIFFER:	Packet sniffer

private: Private data, which can be retrieved using **if_private()**.

DESCRIPTION

Initializes a new netif and installs it, returning the resulting netif. Network device drivers should call this directly, but protocol handlers and packet sniffers should go indirectly through the **if_registervirtual()** function.

The first five arguments are the functions that are associated with netifs. These functions are described in Chapter 4, "Network-Related Kernel Servers."

This function doesn't check any of its arguments, so it always succeeds.

EXAMPLE

```
ifp = if_attach(NULL, myhandler_input, myhandler_output,
    myhandler_getbuf, myhandler_control, name, unit, IFTYPE_IP,
    MYMTU, IFF_BROADCAST, NETIFCLASS_VIRTUAL, ifprivate);
```

SEE ALSO

if_flags(), if_output(), if_init(), if_control(), if_ioctl(), if_getbuf(), if_handle_input()

if_collisions(), if_collisions_set()

SUMMARY Get or set the number of collisions

SYNOPSIS

#include <net/netif.h>

unsigned int **if_collisions**(netif_t *netif*)
void **if_collisions_set**(netif_t *netif*, unsigned int *collisions*)

ARGUMENTS

netif: The module to get or set collision data for.

collisions: The new number of collisions.

DESCRIPTION

if_collisions() gets and **if_collisions_set**() sets the number of collisions encountered. Only the module corresponding to *netif* should call **if_collisions_set**().

EXAMPLE

```
printf("Number of collisions encountered so far:  %d\n",
     if_collisions(netif));
```

if_control(), if_getbuf(), if_init(), if_ioctl(), if_output()

SUMMARY Call one of the functions associated with a netif

SYNOPSIS

#include <net/netif.h>

int **if_control**(netif_t *netif*, const char **command*, void **data*)
netbuf_t **if_getbuf**(netif_t *netif*)
int **if_init**(netif_t *netif*)
int **if_ioctl**(netif_t *netif*, unsigned int *command*, void **data*)
int **if_output**(netif_t *netif*, netbuf_t *packet*, void **address*)

ARGUMENTS

netif: The module whose routine should be called.

packet: The packet to be output.

address: The address to be specified in the call to *netif*'s output function.

command: The control command to be executed.

data: Data specific to the control command.

DESCRIPTION

if_control() calls *netif*'s control routine, **if_getbuf**() calls *netif*'s buffer allocation routine, **if_init**() calls *netif*'s initialization routine, and **if_output**() calls *netif*'s output routine.

We recommend that you don't use **if_ioctl**(); it's provided only for compatibility with UNIX code that operates using **ioctl**. **if_ioctl**() calls *netif*'s control routine.

Except for **if_getbuf**(), these functions return ENXIO if the corresponding function isn't implemented in *netif*; otherwise they return the value returned by the call to the corresponding function. **if_getbuf**() returns NULL if the corresponding function isn't implemented.

EXAMPLE

```
nb = if_getbuf(ifp);
if (nb == NULL)
    return ENOBUFS;
```

SEE ALSO

if_attach()

if_flags(), if_flags_set()

SUMMARY Get or set the flags associated with a netif

SYNOPSIS

#include <net/netif.h>

unsigned int **if_flags**(netif_t *netif*)
void **if_flags_set**(netif_t *netif*, unsigned int *flags*)

ARGUMENTS

netif: The module to get or set flags for.

flags: The new flags.

DESCRIPTION

if_flags() gets and **if_flags_set**() sets the flags associated with *netif*. Only the module corresponding to *netif* should use **if_flags_set**().

Possible flag values are:

IFF_UP:	If true, this interface is working
IFF_BROADCAST:	If true, this interface supports broadcast
IFF_LOOPBACK:	If true, this interface is local only
IFF_POINTTOPOINT:	If true, this is a point-to-point interface

EXAMPLE

```
if_flags_set(ifp, if_flags(ifp) | IFF_UP);
```

SEE ALSO

if_attach()

if_handle_input()

SUMMARY Dispatch an input packet to a protocol handler

SYNOPSIS

#include <net/netif.h>

int **if_handle_input**(netif_t *netif*, netbuf_t *packet*, void **extra*)

ARGUMENTS

netif: This module, which must be a network device driver.

packet: The input packet.

extra: Any extra data that might be needed by the protocol handler.

DESCRIPTION

Call this in a network device driver to have an input packet dispatched to a protocol handler. This routine calls one or more protocol handlers' input routines, passing along the *packet* and *extra* arguments.

This function returns EAFNOSUPPORT if no protocol handler accepts the packet.

EXAMPLE

```
if (nb == 0) {
    printf ("Error:  buffer is null\n");
    goto resetup;
}
else {
    if_handle_input(netif, nb, NULL);
}
```

if_ierrors(), if_ierrors_set(), if_oerrors(), if_oerrors_set()

SUMMARY Get or set the number of input or output errors

SYNOPSIS

#include <net/netif.h>

unsigned int **if_ierrors**(netif_t *netif*)
void **if_ierrors_set**(netif_t *netif*, unsigned int *ierrors*)
unsigned int **if_oerrors**(netif_t *netif*)
void **if_oerrors_set**(netif_t *netif*, unsigned int *oerrors*)

ARGUMENTS

netif: The module for which to access the number of errors.

ierrors: The number of input errors.

oerrors: The number of output errors.

DESCRIPTION

if_ierrors() gets and **if_ierrors_set**() sets the number of input errors encountered. Only the module corresponding to *netif* should call **if_ierrors_set**().

if_oerrors() and **if_oerrors_set**() get and set the number of output errors encountered. Again, only the *netif*'s module should call **if_oerrors_set**().

EXAMPLE

```
error = if_output(lowernetif, nb, (void *)addr);
if (error == 0)
    if_opackets_set(netif, if_opackets(netif) + 1);
else
    if_oerrors_set(netif, if_oerrors(netif) + 1);
return (error);
```

if_ipackets(), if_ipackets_set(), if_opackets(), if_opackets_set()

SUMMARY Get or set the number of packets received or sent

SYNOPSIS

#include <net/netif.h>

unsigned int **if_ipackets**(netif_t *netif*)
void **if_ipackets_set**(netif_t *netif*, unsigned int *ipackets*)
unsigned int **if_opackets**(netif_t *netif*)
void **if_opackets_set**(netif_t *netif*, unsigned int *opackets*)

ARGUMENTS

netif: The module whose packet information is to be accessed.

ipackets: The number of input packets handled by this module since it was loaded.

opackets: The number of packets sent to a lower level by this module since it was loaded.

DESCRIPTION

if_ipackets() gets and **if_ipackets_set()** sets the number of input packets handled. Only the module specified by *netif* should call **if_ipackets_set**().

Similary, **if_opackets**() gets and **if_opackets_set**() sets the number of output packets sent. Only *netif*'s module should call **if_opackets_set**().

EXAMPLE

```
error = if_output(lowernetif, nb, (void *)addr);
if (error == 0) {
    if_opackets_set(netif, if_opackets(netif) + 1);
} else {
    if_oerrors_set(netif, if_oerrors(netif) + 1);
}
return (error);
```

if_mtu(), if_name(), if_private(), if_type(), if_unit()

SUMMARY Get information about a netif

SYNOPSIS

#include <net/netif.h>

unsigned int **if_mtu**(netif_t *netif*)
const char ***if_name**(netif_t *netif*)
void ***if_private**(netif_t *netif*)
const char ***if_type**(netif_t *netif*)
unsigned int **if_unit**(netif_t *netif*)

ARGUMENTS

netif: The netif whose data is being requested.

DESCRIPTION

These functions return the following information about *netif*:

if_mtu():	Its maximum transfer unit (for example, **1500** for Ethernet)
if_name():	Its name (for example,“en”)
if_private():	Its private data
if_type():	Its type string (for example, “10MB Ethernet”)
if_unit():	Its unit number (for example, **0**)

See Chapter 4 for more information on the information that’s associated with a netif. Only the module specified by *netif* should call **if_private**().

EXAMPLE

```
((venip_private_t *)if_private(netif))->lowernetif = realnetif;
```

SEE ALSO

if_flags(), **if_flags_set**(), **if_attach**()

if_registervirtual()

SUMMARY Register a callback function

SYNOPSIS

#include <net/netif.h>

void **if_registervirtual**(if_attach_func_t *attach_func*, void **private*)

ARGUMENTS

attach_func: The callback function.

private: Data to be passed to the callback function.

DESCRIPTION

For use by protocol handlers and packet sniffers. This function registers a callback function and data to be passed to it. See Chapter 4 for more information on implementing a callback function.

EXAMPLE

```
if_registervirtual(myhandler_attach, NULL);
```

inet_queue()

SUMMARY Give an IP input packet to the kernel for processing

SYNOPSIS

#include <net/netif.h>
#include <net/netbuf.h>

void **inet_queue**(netif_t *netif*, netbuf_t *netbuf*)

ARGUMENTS

netif: The protocol handler.

netbuf: The packet to hand over.

DESCRIPTION

IP protocol handlers wishing to hand over their input packets to the kernel for processing should call this function.

You can safely call this function from an interrupt handler, since it doesn't block.

EXAMPLE

```
nb_shrink_top(netbuf, HDRSIZE);
if_ipackets_set(netif, if_ipackets(netif) + 1);
inet_queue(netif, netbuf);
```

nb_alloc(), nb_alloc_wrapper()

SUMMARY Allocate a netbuf or a netbuf wrapper

SYNOPSIS

#include <net/netbuf.h>

netbuf_t **nb_alloc**(unsigned int *size*)
netbuf_t **nb_alloc_wrapper**(void **data*, unsigned int *size*, void *freefunc*(void *), void **freefunc_arg*)

ARGUMENTS

size: The size of the data to be stored.

data: The data to be stored.

freefunc: The function to be called when the netbuf is freed.

freefunc_arg: The argument to be passed to *freefunc*.

DESCRIPTION

nb_alloc() allocates a netbuf containing *size* bytes. It returns the netbuf.

nb_alloc_wrapper() allocates only a wrapper for a netbuf. Use this function when you have already allocated space for the packet. The returned netbuf's original start of data is *data*, and its bottom pointer is initialized to *data* + *size* −1. If **nb_free()** is called on the returned netbuf, *freefunc* will be called with the parameter *freefunc_arg*. *freefunc* can't be null.

Note: Don't call either **nb_alloc()** or **nb_alloc_wrapper()** from an interrupt handler. (Both functions can sleep.)

nb_alloc() and **nb_alloc_wrapper()** return a null pointer if they fail.

EXAMPLE

```
nb = nb_alloc_wrapper((void *)buf, HDR_SIZE + MYMTU,
    mydriver_buf_put, (void *)buf);
```

SEE ALSO

nb_free(), **nb_free_wrapper()**

nb_free(), nb_free_wrapper()

SUMMARY Free a netbuf or only its wrapper

SYNOPSIS

#include <net/netbuf.h>

void **nb_free**(netbuf_t *nb*)
void **nb_free_wrapper**(netbuf_t *nb*)

ARGUMENTS

nb: The netbuf to be freed (or whose wrapper is to be freed).

DESCRIPTION

nb_free() frees the netbuf *nb*. **nb_free_wrapper()** frees only the wrapper of *nb*, leaving its data area intact.

EXAMPLE

```
nb_free_wrapper(nb);
```

SEE ALSO

nb_alloc(), **nb_alloc_wrapper()**

nb_grow_bot(), nb_shrink_bot(), nb_grow_top(), nb_shrink_top()

SUMMARY Change the size of a netbuf

SYNOPSIS

#include <net/netbuf.h>

int **nb_grow_bot**(netbuf_t *nb*, unsigned int *size*)
int **nb_shrink_bot**(netbuf_t *nb*, unsigned int *size*)
int **nb_grow_top**(netbuf_t *nb*, unsigned int *size*)
int **nb_shrink_top**(netbuf_t *nb*, unsigned int *size*)

ARGUMENTS

nb: The netbuf to be affected.

size: The number of bytes to add or delete from the top or bottom pointer.

DESCRIPTION

nb_grow_bot() moves the bottom pointer down. After this call, the data is assumed to end *size* bytes after where it used to end (the data area has effectively grown).

nb_shrink_bot() moves the bottom pointer up, effectively shrinking the data area. After this call, the data is assumed to end *size* bytes before where it used to end.

nb_grow_top() moves the top pointer up, enlarging the data area. After this call, the data is assumed to start *size* bytes before where it used to start.

nb_shrink_top() moves the top pointer down, shrinking the data area. After this call, the data is assumed to start *size* bytes beyond where it used to start.

These functions perform no error checking, so they always succeed and return zero.

Warning: Writing to space outside of the original starting and ending points will cause serious errors, since the extra memory doesn't belong to the netbuf's data section.

EXAMPLE

```
nb_shrink_top(netbuf, HDRSIZE);
if_ipackets_set(netif, if_ipackets(netif) + 1);
inet_queue(netif, netbuf);
```

SEE ALSO

nb_size()

nb_map()

SUMMARY Get a pointer to the data stored in a netbuf

SYNOPSIS

#include <net/netbuf.h>

char ***nb_map**(netbuf_t *nb*)

ARGUMENTS

nb: The netbuf whose data we want.

DESCRIPTION

Returns a pointer to the data stored in *nb*. The pointer is valid only until another **nb_*()** routine is called on *nb*.

This function returns a null pointer if it fails.

EXAMPLE

```
char *map = nb_map(nb);
```

SEE ALSO

nb_read(), **nb_write()**, **nb_size()**

nb_read(), nb_write()

SUMMARY Access data in a netbuf

SYNOPSIS

#include <net/netbuf.h>

int **nb_read**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **target*)
int **nb_write**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **source*)

ARGUMENTS

nb: The netbuf whose data we want to read or write.

offset: The offset of the start of data from the beginning of the netbuf.

size: The number of bytes to be read or written.

target: The place to put the data.

source: The place to read the data from.

DESCRIPTION

nb_read() reads data from *nb* into *target*. It starts at offset *offset* from the starting point in the data and reads *size* bytes into *target*.

nb_write() writes data into *nb*. It starts writing at offset *offset* from the starting point in the data and writes *size* bytes from *source*.

nb_read() and **nb_write**() return zero if the call was successful; otherwise, they return a nonzero value.

EXAMPLE

```
char    buf[MAXTRAILERBUF];

/*
 * Save copy of data.
 */
nb_read(nb, HDRSIZE, offset, &buf);
```

SEE ALSO

nb_map()

nb_size()

SUMMARY Get the size of the data stored in a netbuf

SYNOPSIS

#include <net/netbuf.h>

unsigned int **nb_size**(netbuf_t *nb*)

ARGUMENTS

nb: The netbuf to get the size of.

DESCRIPTION

Returns the size (in bytes) of the data stored by *nb*.

EXAMPLE

```
if (HDRSIZE + offset + size > nb_size(nb)) {
    return (EAFNOSUPPORT);
}
```

SEE ALSO

nb_grow_bot(), **nb_shrink_bot**(), **nb_grow_top**(), **nb_shrink_top**(), **nb_map**()

Appendix A
The Kernel-Server Loader

The kernel-server loader **kern_loader** is the server task that adds loadable kernel servers to the kernel. **kern_loader** works by listening to the ports of known loadable kernel servers. When it intercepts a request for a loadable kernel server, it loads the server and initializes it to respond to this request and subsequent requests.

kern_loader also listens on its own port for requests made through Mach functions called *kernel-server loader functions*. These functions can be used to add and delete known servers, to load servers into the kernel and unload running servers from the kernel, and to get status information. You can use **kl_util** to communicate with **kern_loader** (see Appendix B, "The Kernel-Server Utility"), or you can write your own program using the kernel-server loader functions. These functions are documented in the *Operating System Software* manual.

When invoked, **kern_loader** reads its configuration file, **/etc/kern_loader.conf**. This file contains a list of relocatable object files, one for each kernel server that is to be prepared for loading into the kernel. Here's a sample **kern_loader.conf** file:

```
/usr/lib/kern_loader/Midi/midi_reloc
/usr/lib/kern_loader/NextDimension/NextDimension_reloc
```

Starting kern_loader

The **kern_loader** daemon is called automatically during system startup. If it's killed, you can't normally restart it because the Bootstrap Server won't let any process except **mach_init** register the "server_loader" service. However, if you change a couple of lines in the Bootstrap Server's configuration file and then reboot, the Bootstrap Server will let you reinvoke **kern_loader**. Specifically, you should change the following lines in **/etc/bootstrap.conf**:

```
services NetMessage;
. . .
server "/usr/etc/kern_loader -n" services server_loader;
```

to the following:

```
services NetMessage server_loader;
. . .
server "/usr/etc/kern_loader -n";
```

After you make that change to **/etc/bootstrap.conf** and reboot, you can reinvoke **kern_loader** at any time, as follows:

/usr/etc/kern_loader [**-d**] [**-n**] [**-v**] [*relocatable* ...]

The command-line options are:

-d	Don't detach from the invoking terminal; stay in the foreground.
-n	Don't fork another process to be **kern_loader**. This is necessary in the Bootstrap Server's configuration file because the Bootstrap Server keeps track of all its servers.
-v	Display debugging information.
relocatable ...	The name of one or more relocatable object files to be read (before those listed in **/etc/kern_loader.conf**).

Creating the Relocatable Object File

Your server's relocatable object file must contain certain information: the name of your server, which routines to call to initialize the server, the names of message handling routines, the name of your server's instance variable, and so on. You put this information into the relocatable object file by using **kl_ld** to link your server.

The syntax for using **kl_ld** follows:

kl_ld -n *server_name* **-i** *instance_var* **-l** *load_cmds_file* [**-u** *unload_cmds_file*] [**-d** *loadable_name*] **-o** *output_file input_file ...*

where:

-n *server_name*	Specifies the name of the kernel server. This name is used in calls to the kernel-server loader functions (such as **kern_loader_load_server**()) and in the **kl_util** and **kl_log** command lines.
-i *instance_var*	Specifies the name of the kernel server's instance variable. This variable's structure must start with a field of type **kern_server_t** (defined in the header file **kernserv/kern_server_types.h**).

-l *load_cmds_file*	Specifies the name of the script that contains commands that **kern_loader** must execute when it loads your server. This file is read into the relocatable object file when you create it. If you want to change the load commands, you must recreate the relocatable object file.
-u *unload_cmds_file*	Specifies the name of the script that contains commands that **kern_loader** must execute when it *un*loads your server. Like load commands, unload commands are read into the relocatable object file when you create it. Thus, you must recreate the relocatable object file if you want to change the unload commands.
-d *loadable_name*	Specifies the pathname of the loadable object file that **kern_loader** creates from the relocatable object file. This pathname can be either absolute or relative to the directory containing the relocatable object file. Use this option to make **kern_loader** put the loadable object file in a place where the kernel debugger, KGDB, can easily find and use it.
-o *output_file*	Specifies the name of the relocatable object file that is created. **kern_loader** will later relocate this file against the kernel.
input_file ...	The object files to be linked into the relocatable object file.

The following example shows a makefile that creates a relocatable object file.

Note: On the last line of the command for the "$(NAME)_reloc" target, "$@" refers to "$(NAME)_reloc".

```
NAME=slot
OFILES= slot_server.o slot_handler.o
CFLAGS= -g -DKERNEL -DKERNEL_FEATURES -DMACH
.
.
.
$(NAME)_reloc:  $(OFILES) Load_Commands Unload_Commands
                kl_ld -n $(NAME) -i instance -l Load_Commands \
                    -u Unload_Commands -o $@ $(OFILES)

.c.o:
$(CC) $(CFLAGS) -c $*.c -o $*.o
```

Command Scripts

The load commands script can have the commands described in this section. The script must have at least one of the following commands: HMAP, SMAP, or START.

ADVERTISE	Specifies the name of a port that is to be allocated and advertised with the Network Name Server. When **kern_loader** receives messages on any advertised port, the kernel server will be loaded into the kernel and initialized. As part of the initialization sequence, receive rights for the advertised port are forwarded to the kernel server. The message will then be forwarded by **kern_loader** to the loaded kernel server. Syntax: **ADVERTISE** *port*
CALL	Specifies the name of a function to be called with the specified integer argument as part of the server initialization sequence. If the script has multiple CALL commands, they'll be executed in order. Syntax: **CALL** *function integer*
HMAP	Specifies the mapping of a port to a message handling routine in the kernel server. When **kern_loader** receives a message on this port, it calls the routine with the integer argument you specify. This routine must have a handler interface, as opposed to a server interface (see Chapter 2, "Designing Kernel Servers"). To advertise this port with the Network Name Server, use the ADVERTISE command, above. Syntax: **HMAP** *port_name handler_routine integer*
PORT_DEATH	Specifies a function in the kernel server to be called when a port death message is received on its behalf. Syntax: **PORT_DEATH** *function_name*
SMAP	Specifies the mapping of a port to a message handling routine within the kernel server. When **kern_loader** receives a message on this port, it calls the routine with the integer argument you specify. This routine must have a server interface, as opposed to a handler interface (see Chapter 2). To advertise this port with the Network Name Server, use the ADVERTISE command, above. Syntax: **SMAP** *port_name server_routine integer*
START	Causes the kernel server to be started immediately, rather than waiting for a message to be received on one of its advertised ports. This is most appropriate for kernel servers that don't listen on any ports, or are wired into kernel data structures for non-server-style access. Syntax: **START**

WIRE	Causes the text and data of the loaded kernel server to be wired down (memory-resident), making the kernel server immune from unexpected page faults. You must use WIRE if any part of your kernel server can be called from an interrupt handler. If you use WIRE, your kernel server is wired down before any other load commands are executed. Syntax: **WIRE**

Here's an example of a load commands script.

```
CALL  slot_init 0

PORT_DEATH  slot_port_death

# Associate ports with proc/arg
SMAP  slot0 slot_msg 0
SMAP  slot2 slot_msg 1
SMAP  slot4 slot_msg 2
SMAP  slot6 slot_msg 3

# Server contains interrupt handler code, and so must be wired down
WIRE

# Start this server up immediately
START
```

The unload commands script can have only CALL commands:

CALL	Specifies the name of a function to be called as part of server shutdown. The function will be passed the specified integer. Syntax: **CALL** *function integer*

Here's an example of an unload commands script.

```
# Termination

CALL   slot_signoff  0
```

Appendix B
The Kernel-Server Utility

The kernel-server utility **/usr/etc/kl_util** lets you communicate with the kernel-server loader. Various options allow you to query the kernel loader for the status of all registered kernel servers, load a kernel server into the kernel, and remove one or more kernel servers from the kernel.

The command-line options to **kl_util** are as follows:

-a *server_reloc_file_name ...*
: Causes **kern_loader** to allocate resources for the specified kernel server or servers. Each added server will have kernel space allocated for it and will be initialized to load at that location when referenced.

-A
: Causes **kern_loader** to shut down; all existing kernel servers are unloaded and deallocated, and the running **kern_loader** task exits.

-d *server_name ...*
: Causes **kern_loader** to deallocate the specified kernel server or servers; all physical and virtual resources associated with the kernel server are freed.

-l *server_name ...*
: Causes **kern_loader** to load the specified kernel server or servers into the kernel. If you don't use this option, loading is normally done either when the kernel server is allocated (if START is specified in the load commands) or when it receives its first message.

-L
: Causes **kl_util** not to terminate at the end of its operation, so that further **kern_loader** activity can be monitored. As long as **kl_util** is running, anything logged by **kern_loader** is displayed.

-r
: Causes **kern_loader** to deallocate all its servers and set itself up from scratch by rereading its configuration file. This is similar to specifying the **-A** option and then restarting **kern_loader**, except that **kern_loader** never actually exits.

-s [*server_name ...*]

Causes **kern_loader** to return information about the status of registered kernel servers. If a server name isn't specified, a list of all known servers is displayed. If a server name is specified, detailed information about that server is displayed.

The following example shows the status of the MIDI driver:

```
# /usr/etc/kl_util -s midi
SERVER: midi
RELOCATABLE: /usr/lib/kern_loader/Midi/midi_reloc
STATUS: Allocated at address 0x10e3a000 for 0x8000 bytes
PORTS: midi0(advertised) midi1(advertised)
#
```

The information returned includes the name of each registered kernel server, a status of either *loaded* (indicating that the kernel server is loaded and running in the kernel) or *allocated* (indicating that the kernel server space has been allocated in the kernel, and that the relocatable object file has been relocated at the allocated address but has not yet been loaded into the kernel). The output also includes the name of each of the kernel server's ports.

-u *server_name ...*

Causes **kern_loader** to unload the specified kernel server or servers. (Loaded kernel servers remain in the kernel until they're explicitly unloaded.) Unloading the server causes any wired pages to be unwired; thus, this can be used as a mechanism to free up resources in the system when the server is no longer needed.

Appendix C
The Kernel-Server Log Command

You can use the kernel-server log command, **kl_log**, to see log messages from a loadable kernel server. If you wish, you can instead write your own program that calls the **kern_loader_log_level**() and **kern_loader_get_log**() functions to get log messages. **kern_loader_log_level**() and **kern_loader_get_log**() are discussed in the *Operating System Software* manual.

You must be superuser to call **kl_log**. It has the following syntax:

/usr/etc/kl_log [**-l** *log_level*] *server_name*

where:

server_name	Specifies the loadable kernel server for which you're getting or setting log information. This server must be loaded already.
-l *log_level*	Specifies the priority of messages that should be kept. By default, the log level is zero, and no log messages are printed. By setting *log_level* to a positive value, you ensure that log messages from the server that have a priority equal to or greater than *log_level* will be printed to **stdout**.

You might use **kl_log** as follows:

```
slave# kl_log -l 1 mydriver&
.
.
.
slave# kl_log -l 0 mydriver
slave# jobs
[1]  + Running              kl_log -l 1 mydriver
slave# kill %1
slave#
```

Before you stop collecting messages from a kernel server, you should shut off logging by setting its log level to zero. If you don't set the log level to zero, log messages will accumulate even though no process is collecting the messages.

Appendix D
The Kernel Debugger

This appendix describes how to debug your server using the kernel debugger, KGDB. KGDB is a superset of GDB; it has all the GDB commands, plus a few that are designed specifically for debugging kernels.

With KGDB, you can debug every function in your server that's called after your server is loaded. However, you can't debug functions that are called when **kern_loader** is initializing your server.

Besides any hardware needed for your server, KGDB also requires:

- Two NeXT computers: the test computer (on which your server will run) and another computer.

- Either working network connections for both machines or an RS-422 cable to connect the two computers. If you have a network, then the test computer must have a legitimate hostname that the other machine can find with NetInfo™. If you don't have a network, then you must use the Macintosh®-to-ImageWriter® II cable, null-modem style DIN-8 to DIN-8 (Businessland® order # 200-66696). Make sure the cable has identical round interfaces on both ends, and make sure that it's null-modem style (not straight through).

Warning: If you don't have a network and your computers have different CPU chips (one has a 68030 and the other has a 68040), you must use a custom serial cable. The **zs** UNIX manual page describes this cable.

KGDB runs on one computer (the *master*), debugging the kernel on the test computer (the *slave*). Keep in mind that no one can depend on the slave computer working all the time, since you'll need to reboot it often. For example, you should not use the slave as an NFS® server. Remember, too, that the operating system for the slave computer will often be halted, so you should copy any files you might need from the slave onto another computer.

Once you have the hardware, follow these steps to debug your kernel (as described in detail below):

1. Set up the hardware.
2. Put the appropriate files where KGDB can find them.
3. Start up and initialize KGDB.
4. Debug with KGDB.

Setting Up the Hardware

Set up the computers as you normally would, connecting them to the network if you have one. You'll need access to both keyboards, so put the computers close together.

If you're connecting these computers with a serial cable, first make sure there's no **getty()** running on either computer's serial port A. Next, plug the RS-422 cable into serial port A of both computers.

Once you've set up the hardware, boot both systems. Use the **-p** option to the **boot** command on the slave computer so the Panic window will stay up.

Setting Up the Files

First you must decide how you're going to keep your server files in synch between the master and slave computers. There are two considerations here: keeping the files in synch and keeping disk space on the slave to a minimum (to avoid long disk checks after panics).

You can either make the slave the NetBoot client of the master, or use NFS to mount the directory containing the relocatable object file on the slave computer. Using NFS and NetBoot is covered in the *Network and System Administration* manual.

If you can't use a network, you must use a removable disk to copy the appropriate files from the development computer to the slave computer (and, if you're using a third machine for development, to the master computer). You must copy the files over whenever they change.

Files Needed by the Slave Computer

The slave computer needs only whatever files are required by **kern_loader**. Usually, this is just your server's relocatable object file. You must create the relocatable object file by compiling with the **-g** option so that it contains debugging information. Avoid using the **-O** option, since optimization can make variable values appear incorrect.

Files Needed by the Master Computer

The master computer needs access to the following:

- The directory that contains the source files for your server.

- Your server's loadable object file. This file is produced by **kern_loader** on the slave computer when your server is allocated. See Appendix A, "The Kernel-Server Loader," for information on how to specify the location of this loadable object file.

- A file that contains the same version of the kernel as the one that the slave is running. If the master and the slave are running the same version of the kernel, then you can use the **/mach** file on the master. You can check the version by searching for "mk-" in **/usr/adm/messages**.

Before you go on to the next step, write down the full pathnames for the master computer's server source directory and loadable object file. You'll need to supply these pathnames to KGDB later.

Starting Up KGDB

On the master computer:

1. Become **root** in a Shell or Terminal window.

```
master> su
Password:
```

2. Change to the directory containing the kernel file that's the same version as the one running on the slave.

```
master:1# cd /
```

3. Start KGDB.

```
master:2# kgdb mach
```

You'll see some messages, ending with something like:

```
(no debugging symbols found)...done.
Type "help" for a list of commands.
(gdb)
```

4. Unless the slave computer was booted with the **-p** option, set a breakpoint for **panic()**. Setting this breakpoint ensures that you'll be able to use KGDB's **backtrace** command to see what caused the panic.

```
(gdb) break panic
Breakpoint 1 at 0x4009da8
(gdb)
```

5. Establish the master computer's control over the slave, using the **kattach** command. If your computers are connected over the network, then enter "**kattach** *hostname*", where *hostname* is the name of the slave machine. If your computers are connected with a serial cable, then enter "**kattach /dev/ttya**".

```
(gdb) kattach slave
Attaching to running kernel
Connecting to slave...
```

Because the slave's kernel is running, you won't see another "(gdb)" prompt until the slave's kernel hits a breakpoint or you enter the NMI mini-monitor on the slave (and, for serial connections, enter the **gdb** command).

On the slave computer:

6. Load the server, if it isn't already running.

7. If the loadable object file isn't currently accessible to the master computer, copy it over to the master computer.

8. Wait for a few seconds after the **kattach** command. Then get into the NMI mini-monitor by pressing Command-Command-~ (hold down both Command keys and press the key at the upper left of the numeric keypad). If your systems are connected by a serial cable, you must also enter **gdb** at the NMI mini-monitor window.

```
nmi> gdb
```
(only for systems connected with a serial cable)

The slave computer is now frozen because its kernel is stopped.

On the master computer:

9. Now that you have the "(gdb)" prompt back, you can bring the symbol information from your loadable file into KGDB. The amount of time KGDB takes to read the symbol information depends on the size and complexity of your file. You must use the **add-file** command, followed by the full pathname of the loadable file, followed by **0**. For example:

```
Program received signal 5, Trace/BPT trap
0x4053ad6 in kdbg_connect ()
(gdb) add-file /me/Drivers/slot_loadable 0
add symbol table from filename "/me/Drivers/slot_loadable" at
text_addr = 0x0
(y or n) y
Reading symbol data from /me/Drivers/slot_loadable...done.
(gdb)
```

If you had any trouble adding your server to KGDB, make sure **kl_util -s** *servername* on the slave shows your server as "Loaded." If not, load it.

10. Finally, tell KGDB where your source files are with the **dir** or **idir** command. For example:

```
(gdb) dir /me/Drivers/slotSrc
Source directories searched: /me/Drivers/slotSrc
(gdb)
```

Debugging with KGDB

You're now ready to set breakpoints and debug your code. When you're ready to continue running the kernel, use the **cont** (continue) command in KGDB. If it won't continue, make sure your server is loaded, not just allocated, on the slave computer.

Warning: Never use the **run** command in KGDB. It causes unpredictable behavior.

If you want to get a "(gdb)" prompt on the master, you must stop the slave's kernel by generating an NMI (press Command-Command-~) and entering **gdb** at the "nmi>" prompt in the NMI mini-monitor. Even if your machines are connected over a network, and thus you didn't have to enter **gdb** in step 8 above, you need to use the **gdb** command from now on.

When you change or reload your server, you don't have to exit KGDB. Instead, just use **add-file** (see step 9, above) to load the new loadable object file into KGDB. If the source has changed, make sure that the new source files are in the master's source directory.

Ending the Debugging Session

To remove KGDB from a running kernel, follow the steps below. You can't use the GDB **quit** command because the kernel on the slave computer needs to recover from whatever state KGDB has left it in.

1. If you don't have a "(gdb)" prompt on the master computer, get one by generating an NMI at the slave computer (press Command-Command-~) and entering **gdb** at the "nmi>" prompt.

At the master computer:

2. Delete all the breakpoints you've set.

```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb)
```

3. Continue the kernel's execution, then stop KGDB by typing Control-Z.

```
(gdb) cont
Continuing.
<Control-Z>
Stopped.
master:3#
```

4. Kill the KGDB process.

```
master:3# jobs
  [1] +Stopped /me/Apps/kgdb mach
master:4# kill %1
master:5#
```

Appendix E
The ROM Monitor and NMI Mini-Monitor

Most of the commands in the ROM monitor and NMI mini-monitor are discussed in the *Network and System Administration* manual; some other useful commands are covered in other chapters of this manual. However, a few commands are of interest to only a few hardware or device driver developers. These rarely used commands are described in this appendix.

See Chapter 3, "Testing and Debugging Kernel Servers"; Appendix D, "The Kernel Debugger"; and the *Network and System Administration* manual for more information on using the ROM monitor and NMI mini-monitor.

ROM Monitor Commands

Most ROM monitor commands display the current value of a register or parameter and prompt for a new value with a question mark. To enter a new value, type the value and press the Return key. To leave the current value unchanged and skip to the next one (if any), just press the Return key. Type a period to exit the command and leave the current value unchanged.

Numeric values are usually in hexadecimal (base 16) notation if they represent a memory address or data value. The **R** (radix) command can be used to change the input base/radix.

The values of registers' bit fields are displayed symbolically between angle brackets < > in addition to the numeric value (for example, "2700<trace=0,s,ipl=7>"). For more information on 68040 registers, see Motorola's *MC68040 User's Manual.*

Open Address Register

The command

a [*n*]

accesses the contents of address registers a0 through a7 of the 680x0 processor.

Example:

```
NeXT> a
a0: 00001234? 5678          Change a0 from 0x1234 to 0x5678
a1: 00000000?               Don't change a1
a2: 00000001? .             Type a period to exit

NeXT> a0
a0: 00001234? 5678          Change a0 from 0x1234 to 0x5678
NeXT>
```

Open Data Register

The command

d [*n*]

accesses the contents of data registers d0 through d7 of the 680x0 processor.

Example:

```
NeXT> d
d0: 00001234? 5678          Change d0 from 0x1234 to 0x5678
d1: 00000000?               Don't change d1
d2: 00000001? .             Type a period to exit

NeXT> d0
d0: 00001234? 5678          Change d0 from 0x1234 to 0x5678
NeXT>
```

Open Processor Register

The command

r [*regname*]

accesses the contents of the 680x0 processor registers. Possible values for *regname* are:

pc	Program counter
sr	Status register
usp	User stack pointer
isp	Interrupt stack pointer
msp	Master stack pointer
vbr	Vector base register
sfc	Source function code
dfc	Destination function code
cacr	Cache control register
caar	Cache address register (68030 only)

If you don't specify *regname*, all the processor registers are opened in the order they're listed above.

Open System Register

The command

s [*systemreg*]

accesses the contents of the system registers. Possible values for *systemreg* are:

intrstat	Interrupt status register
intrmask	Interrupt mask register
scr1	System control register #1
scr2	System control register #2

If you don't specify *systemreg*, all of the system registers are opened in the order they're listed above.

Examine Memory Locations

The command

e [*lwb*] [*addrlist*] [*format*]

lets you examine particular locations in memory.

[*lwb*]	Specify **l**, **w**, or **b** to select long, word, or byte length. The default is long.
[*addrlist*]	This argument specifies the starting address or list of addresses to cyclically examine. If you don't specify a value for *addrlist*, the command uses the most recent value of *addrlist*.
[*format*]	This argument controls how the value is displayed. It can be any of the standard format types supported by the C language **printf()** library routine. The default, **%x**, displays a number in hexadecimal.

Example 1:

`NeXT> e 4000000`	Examine the long at memory location 0x4000000
`4000000: 0? 12345678`	See that its value is 0x0; deposit the value 0x12345678
`4000004: 0? .`	Type a period to exit

Example 2:

`NeXT> e 4000000 4000000`	Repeatedly examine memory location 0x4000000
`4000000: 0? 12345678`	See that its value is 0x0; deposit the value 0x12345678
`4000000: 12345678? .`	See that its value is 0x12345678; type a period to exit

Open Function Code

The command

S [*fcode*]

lets you inspect or modify the 680x0 function code (address space) used with the **e** (examine) command. The default code is 5 (supervisor data space).

Possible values for *fcode* are:

0	Undefined, reserved
1	User data space (UD)
2	User program space (UP)
3	Undefined, reserved
4	Undefined, reserved
5	Supervisor data space (SD)
6	Supervisor program space (SP)
7	CPU space

Example:

```
NeXT> S
Function code 5 (SD)          Current function code is 5
NeXT> S 6
Function code 6 (SP)          Change function code to 6
```

Set Input Radix

The command

R [*radix*]

lets you set the input radix. The default value for the input radix is 16. Any numbers you type in are interpreted in the base of the input radix (for example, input radix 16 means numbers are interpreted as base 16). If you don't specify *radix*, the current default input radix is displayed.

NMI Mini-Monitor Commands

The following NMI mini-monitor command lets you set kernel flags. However, this feature isn't currently useful for developers outside of NeXT.

Set or Examine Any Kernel Flag

The command

flag=[*value*]

lets you examine or modify internal system flags (most internal system flags are of interest only to system developers at NeXT).

Example:

```
nmi> debug=
kernel flags debug = 0x0      The debug flag has the value 0
nmi> debug=2
kernel flag = 0x2             Change the value of the debug flag to 2
```

Appendix F
Summary of Kernel Functions

General Functions

This section contains a summary of the general-purpose kernel functions, which are described in detail in Chapter 5, "C Functions."

Time Functions

Busy-wait for a certain amount of time:

void	**DELAY**(unsigned int *usecs*)

Get the current time:

void	**microboot**(struct timeval **tvp*)
void	**microtime**(struct timeval **tvp*)

Schedule or unschedule a function to be called later:

void	**us_abstimeout**(int (**function*)(), vm_address_t *arg*, struct timeval **tvp*, int *priority*)
void	**us_timeout**(int (**function*)(), vm_address_t *arg*, struct timeval **tvp*, int *priority*)
boolean_t	**us_untimeout**(int (**function*)(), vm_address_t *arg*)

Memory Functions

Make addresses pageable or memory-resident:

kern_return_t	**kern_serv_unwire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)
kern_return_t	**kern_serv_wire_range**(kern_server_t **ksp*, vm_address_t *address*, vm_size_t *size*)

Copy or initialize data:

void	**bcopy**(void **from*, void **to*, int *length*)
void	**bzero**(void **address*, int *length*)
int	**copyin**(void **from*, void **to*, int *length*)
int	**copyout**(void **from*, void **to*, int *length*)

Allocate or free memory:

void *	**kalloc**(int *size*)
void	**kfree**(void **address*, int *size*)
void *	**kget**(int *size*)

Critical Section and Synchronization Functions

Use read and write locks:

lock_t	**lock_alloc**()
void	**lock_free**(lock_t *lock*)
void	**lock_done**(lock_t *lock*)
void	**lock_init**(lock_t *lock*, boolean_t *can_sleep*)
void	**lock_read**(lock_t *lock*)
void	**lock_write**(lock_t *lock*)

Use simple, non-sleeping locks:

void	**simple_lock**(simple_lock_t *lock*)
simple_lock_t	**simple_lock_alloc**()
void	**simple_lock_free**(simple_lock_t *lock*)
void	**simple_lock_init**(simple_lock_t *lock*)
void	**simple_unlock**(simple_lock_t *lock*)

Cause a thread to sleep or wakeup:

void	**assert_wait**(int *event*, boolean_t *interruptible*)
void	**biodone**(struct buf **bp*)
void	**biowait**(struct buf **bp*)
void	**clear_wait**(thread_t *thread*, int *result*, boolean_t *interrupt_only*)
void	**thread_block**()
void	**thread_set_timeout**(int *ticks*)
void	**thread_sleep**(int *event*, simple_lock_t *lock*, boolean_t *interruptible*)
void	**thread_wakeup**(int *event*)

General Task and Thread Functions

Get information about this thread or task:

task_t	**current_task**()
int	**thread_wait_result**()

Create or kill a thread:

thread_t	**kernel_thread**(task_t *task*, void (**start*)())
void	**thread_halt_self**()

Port and Message Functions

Request notification messages, such as port death notification:

kern_return_t	**kern_serv_notify**(kern_server_t **ksp*, port_t *reply_port*, port_t *request_port*)

Get or set information about this server's ports:

port_t	**kern_serv_bootstrap_port**(kern_server_t **ksp*)
port_t	**kern_serv_local_port**(kern_server_t **ksp*)
port_t	**kern_serv_notify_port**(kern_server_t **ksp*)
void	**kern_serv_port_gone**(kern_server_t **ksp*, port_name_t *port*)
kern_return_t	**kern_serv_port_proc**(kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg*)
kern_return_t	**kern_serv_port_serv**(kern_server_t **ksp*, port_all_t *port*, port_map_proc_t *function*, int *arg*)
port_set_name_t	**kern_serv_port_set**(kern_server_t **ksp*)

Hardware Interface Functions

Set up or remove an interrupt handler:

int	**install_polled_intr**(int *which*, int (**my_intr*)())
int	**uninstall_polled_intr**(int *which*, int (**my_intr*)())

Get or test a virtual address that corresponds to a hardware address:

caddr_t	**map_addr**(caddr_t *address*, int *size*)
int	**probe_rb**(void **address*)

Change or determine the processor level:

int	**curipl**()
int	**spl0**(), **spl1**(), **spl2**(), **spl3**(), **spl4**(), **spl5**(), **spl6**(), **spl7**()
	splx(int *priority*)

Logging and Debugging Functions

Kill the loadable kernel server:

void	**ASSERT**(int *expression*)
kern_return_t	**kern_serv_panic**(port_t **bootstrap_port*, panic_msg_t *message*)
void	**panic**(char **string*)

Log a message:

void	**kern_serv_log**(kern_server_t **ksp*, int *log_level*, char **format*, *arg1*, ..., *arg5*)
int	**log**(int *level*, char **format*, *arg*, ...)
int	**printf**(char **format* [, *arg1*, ...])

Miscellaneous Functions

Modify or inspect a string:

int	**sprintf**(char **string*, char **format* [, *arg1*, ...])
char *	**strcat**(char **string1*, char **string2*)
int	**strcmp**(char **string1*, char **string2*)
char *	**strcpy**(char **to*, char **from*)
int	**strlen**(char **string*)

In a UNIX-style server, determine whether the user has root privileges:

int	**suser**()

Call a function from the main thread:

kern_return_t	**kern_serv_callout**(kern_server_t **ksp*, void (**func*)(void *), void **arg*)

Network Functions

This section contains a summary of the network-specific kernel functions, which are described in detail in Chapter 5, "C Functions." A general discussion of networking drivers and protocols is in Chapter 4, "Network-Related Kernel Servers."

Netif Functions

To use these functions, you need to include the header file **<net/netif.h>**.

Initialize and install a new netif:

netif_t	**if_attach**(if_init_func_t *init_func*, if_input_func_t *input_func*, if_output_func_t *output_func*, if_getbuf_func_t *getbuf_func*, if_control_func_t *control_func*, const char **name*, unsigned int *unit*, const char **type*, unsigned int *mtu*, unsigned int *flags*, netif_class_t *class*, void **private*)
void	**if_registervirtual**(if_attach_func_t *attach_func*, void **private*)

Get or set data for a netif:

unsigned int	**if_collisions**(netif_t *netif*)
void	**if_collisions_set**(netif_t *netif*, unsigned int *collisions*)
unsigned int	**if_flags**(netif_t *netif*)
void	**if_flags_set**(netif_t *netif*, unsigned int *flags*)
unsigned int	**if_ierrors**(netif_t *netif*)
void	**if_ierrors_set**(netif_t *netif*, unsigned int *ierrors*)
unsigned int	**if_oerrors**(netif_t *netif*)
void	**if_oerrors_set**(netif_t *netif*, unsigned int *oerrors*)
unsigned int	**if_ipackets**(netif_t *netif*)
void	**if_ipackets_set**(netif_t *netif*, unsigned int *ipackets*)
unsigned int	**if_opackets**(netif_t *netif*)

void	**if_opackets_set**(netif_t *netif*, unsigned int *opackets*)
unsigned int	**if_mtu**(netif_t *netif*)
const char *	**if_name**(netif_t *netif*)
void *	**if_private**(netif_t *netif*)
const char *	**if_type**(netif_t *netif*)
unsigned int	**if_unit**(netif_t *netif*)

Call a netif's function:

int	**if_control**(netif_t *netif*, const char **command*, void **data*)
netbuf_t	**if_getbuf**(netif_t *netif*)
int	**if_init**(netif_t *netif*)
int	**if_ioctl**(netif_t *netif*, unsigned int *command*, void **data*)
int	**if_output**(netif_t *netif*, netbuf_t *packet*, void **address*)

Dispatch a packet to a protocol handler:

int	**if_handle_input**(netif_t *netif*, netbuf_t *packet*, void **extra*)

Netbuf Functions

You must include **<net/netbuf.h>** when you use these functions.

Allocate or free a netbuf or its wrapper:

netbuf_t	**nb_alloc**(unsigned int *size*)
netbuf_t	**nb_alloc_wrapper**(void **data*, unsigned int *size*, void *freefunc*(void *), void **freefunc_arg*)
void	**nb_free**(netbuf_t *nb*)
void	**nb_free_wrapper**(netbuf_t *nb*)

Change the size of a netbuf:

int	**nb_grow_bot**(netbuf_t *nb*, unsigned int *size*)
int	**nb_shrink_bot**(netbuf_t *nb*, unsigned int *size*)
int	**nb_grow_top**(netbuf_t *nb*, unsigned int *size*)
int	**nb_shrink_top**(netbuf_t *nb*, unsigned int *size*)

Access the data in a netbuf:

char *	**nb_map**(netbuf_t *nb*)
int	**nb_read**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **target*)
int	**nb_write**(netbuf_t *nb*, unsigned int *offset*, unsigned int *size*, void **source*)
unsigned int	**nb_size**(netbuf_t *nb*)

Miscellaneous Functions

For the host-network conversion functions, you need to include **<netinet/in.h>**. For **inet_queue**(), you must include both **<net/netif.h>** and **<net/netbuf.h>**.

Convert values between host and network byte order:

u_long	**htonl**(u_long *hostlong*)
u_short	**htons**(u_short *hostshort*)
u_long	**ntohl**(u_long *netlong*)
u_short	**ntohs**(u_short *netshort*)

Give an IP input packet to the kernel for processing:

void	**inet_queue**(netif_t *netif*, netbuf_t *netbuf*)

Index

NeXT Computer, Inc.
900 Chesapeake Drive
Redwood City, CA 94063

Printed in U.S.A.
1386.00
12/90