3

Using Loadable Kernel Servers

This chapter discusses how to use the kernel-server loader functions to interact with *loadable kernel servers*. Loadable kernel servers are modules, such as device drivers and network protocols, that can be added to the NeXT Mach kernel.   One example of interaction with a loadable kernel server is using the function **kern_loader_load_server()** to load a loadable kernel server.   Another example is using the function **kern_loader_server_list()** to get a list of all loadable kernel servers that are either loaded or prepared for loading (allocated).

The following section gives more information on loadable kernel servers and on the kernel-server loader, **kern_loader**.   The next section gives some examples of using the kernel-server loader functions.   Each kernel-server loader function is described in detail in Chapter 4, ªMach Functions.º

For more information about loadable kernel servers, see Part 2 of this manual, ªWriting Loadable Kernel Servers.º Part 2 also has information about the kernel-server utility, **kl_util**, which is a command-line interface to many of the functions described in this€chapter.

How Loadable Kernel Servers Work

The kernel-server loader is a server task that's automatically called during system startup.   When started, it reads a list of loadable kernel servers out of its configuration file, **/etc/kern_loader.conf**, and allocates these servers.

A loadable kernel server is a module that's loaded into the kernel after the system has been booted.   Because loadable kernel servers are the only way to add kernel functionality without recompiling the whole kernel, they're the only way for anyone outside of NeXT to write kernel-level device drivers and network protocols.   However, third parties aren't the only ones to use loadable kernel serversÐNeXT uses them for drivers of devices that many people won't have.

For example, the graphics tablet driver is a loadable kernel server that is loaded by the application **/NextAdmin/InstallTablet**.   Having a loadable tablet driver is advantageous because performance on NeXT computers that don't have a graphics tablet (the majority of NeXT computers) is better than if the tablet driver were always in the kernel.

Loadable kernel servers can have three states:

· AllocatedÐThe kernel-server loader (**kern_loader**) has allocated space and resources for the loadable kernel server and is listening for Mach messages to its ports.   However, the server isn't currently loaded into the kernel.

· LoadedÐThe loadable kernel server is running.

· UnallocatedÐThe kernel-server loader has no space or other resources allocated for the loadable kernel server.

Not all loadable kernel servers stay in the allocated state when they're initialized.   Servers that don't use Mach messages, for example, are loaded immediately.   Most message-based servers, however, stay in the allocated state until the kernel-server loader receives either a message for the server or a request such as

kern_loader_load_server() that tells it to load the server.

Each loadable kernel server stays loaded until the kernel-server loader either shuts down or receives a request to unload or delete the server (such as **kern_loader_unload_server()**).

See Chapter 5, ªOverview of Loadable Kernel€Servers,º for more information on loadable kernel servers and on using **kern_loader**.

Overview of Kernel-Server Loader Functions

This section describes the use of the kernel-server loader functions. See the ªKernel-Server Loader Functionsº section of Chapter 4 for more information on each of the functions.

The kernel-server loader functions are broken into two groupsÐthose that deal with a single loadable kernel server, and those that deal with the kernel-server loader itself. Two more functions help you print error messages: **kern_loader_error()** and **kern_loader_error_string()**.

Before you can call any other kernel-server loader function, you must call **kern_loader_look_up()** to obtain the port of the kernel-server loader. You must provide this port as an argument to all of the following function calls. A similar argument, returned by **kern_loader_server_com_port()**, is required only for calls to functions that deal with a server's message logging.

Use **kern_loader_add_server()** to cause a loadable kernel server to be allocated. If the server starts automatically, then it will be loaded into the kernel; otherwise, you can send a message to the server or call **kern_loader_load_server()** to load the server into the kernel. To remove a loadable kernel server from the kernel, use **kern_loader_unload_server()** (to leave the server in the allocated state) or **kern_loader_delete_server()** (to deallocate kernel-server loader resources for the server).

For each loadable kernel server, logging is off by default. To get status messages from a particular loadable kernel server, use **kern_loader_log_level()** to turn the server's logging on and **kern_loader_get_log()** to get the next log message. You might want to turn logging off (by again using **kern_loader_log_level()**) before you stop collecting log messages, since messages continue to be logged and take system space even when no one requests them.

You can get detailed information about the state of a particular server by calling **kern_loader_server_info()**.

Use **kern_loader_status_port()** to register a port where log messages from the kernel-server loader should be sent. These messages usually reflect changes in the state of one or more loadable kernel servers. You can get a list of all the servers that the kernel-server loader knows about by calling **kern_loader_server_list()**. Use **kern_loader_abort()** to shut down or reconfigure the kernel-server loader. Use **kern_loader_ping()** to make sure either that the kernel-server loader is responding normally to messages, or that all outstanding status messages have been sent.

Functions for Asynchronous Messages

Three of the kernel-server loader functions don't immediately return information. Instead, these three functions tell the kernel-server loader to send asynchronous reply messages that contain the information. Whenever you call one of these functions, you must supply the code necessary to handle the kernel-server loader's reply message. The following table shows the three asynchronous kernel-server loader functions and their corresponding user-written functions.

Asynchronous Function	User-Written Function
kern_loader_ping()	*ping_func()*
kern_loader_get_log()	*log_data_func()*
kern_loader_status_port()	*string_func()*

This section describes how to handle asynchronous reply messages from the kernel-server loader. First it describes the code that all three of the asynchronous functions require in your program. Then it describes how to implement the handler necessary for each of the functions.

Common Code for Handling Reply Messages

If your program calls a kernel-server loader function that sends an asynchronous reply message, then your program must follow these steps to handle reply messages:

1. Allocate a port on which to receive messages from the kernel-server loader.
2. Call the asynchronous function, passing as data the receiving port.
3. Listen to the receiving port (often in a separate thread).
4. After receiving a message on the port, call **kern_loader_reply_handler()**.
5. Take care of the reply message in a handling function, which is called by **kern_loader_reply_handler()**.

You must write the handling function that's called by **kern_loader_reply_handler()**. You must also create a structure that specifies which handling functions exist; you pass a pointer to this structure to **kern_loader_reply_handler()** every time you call it. The structure is of type **kern_loader_reply_t**, which is defined in the header file **kernserv/kern_loader_reply_handler.h** as the following:

```
typedef struct kern_loader_reply {
    void         *arg;        /* argument to pass to function */
    msg_timeout_t timeout;    /* timeout for RPC return msg_send */
    kern_return_t (*string)(  /* kern_loader_status_port() function */
                void         *arg,
                printf_data_t string,
                unsigned int  string_count,
                int           level);
    kern_return_t (*ping)(    /* kern_loader_ping() function */
                void      *arg,
                int        id);
    kern_return_t (*log_data)(/* kern_loader_get_log() function */
                void         *arg,
                printf_data_t log_data,
                unsigned int  log_data_count);
} kern_loader_reply_t;
```

The following example calls one of the asynchronous kernel-server loader functions, **kern_loader_status_port()**. The handler for the reply message is called **print_string()**, and is specified to the kernel-server loader using the structure **reply_handlers**.

```
#import <mach/mach.h>
#import <mach/mach_error.h>
#import <kernserv/kern_loader_types.h>
#import <kernserv/kern_loader.h>
#import <kernserv/kern_loader_reply_handler.h>
#import <kernserv/kern_loader_error.h>
#import <mach/cthreads.h>
#import <libc.h>
#import <stdio.h>

void receive_thread(port_name_t port);
kern_return_t print_string(void *arg, printf_data_t string,
    unsigned int string_count, int level);

void main()
{
    kern_return_t   r;
    port_name_t     status_port, kl_port;
    r = kern_loader_look_up(&kl_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't find kernel loader", r);
        exit(1);
    }

    r = port_allocate(task_self(), &status_port);
    if (r != KERN_SUCCESS) {
        mach_error("kl_util: can't allocate reply port", r);
        exit(1);
    }

    /* Get generic status messages on this port. */
    r = kern_loader_status_port(kl_port, status_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("Couldn't specify status port", r);
            exit(1);
    }

    /* Create a thread to listen on status_port. */
    cthread_detach(cthread_fork((cthread_fn_t)receive_thread,
        (any_t)status_port));

    /*
     * Sleep for a while so we can enter kl_util commands at a shell
     * window.  The output of all commands (except status lines from
     * kl_util -s) will show up in both the window that's running this
     * program and in the window that's running kl_util.  (kl_util
     * also has a status port registered.)
     */
    sleep(30);
    exit(0);
}

kern_loader_reply_t reply_handlers = {
    0,                  /* argument to pass to function */
    0,                  /* timeout for rpc return msg_send */
    print_string,       /* string function */
    0,                  /* reply_ping function */
    0                   /* log_data function */
};

void receive_thread(port_name_t port)
{
    char           msg_buf[kern_loader_replyMaxRequestSize];
    msg_header_t *msg = (msg_header_t *)msg_buf;
    kern_return_t r;

    /* message handling loop */
    while (TRUE) {
        /* Receive the next message in the queue. */
        msg->msg_size = kern_loader_replyMaxRequestSize;
        msg->msg_local_port = port;
        r = msg_receive(msg, MSG_OPTION_NONE, 0);
        if (r != KERN_SUCCESS)
            break;

        /* Handle the message we just received. */
        kern_loader_reply_handler(msg, &reply_handlers);
    }

    /* We get here only if msg_receive returned an error. */
    mach_error("receive_thread", r);
    exit(1);
}

/*
```

```
 * This function is called by kern_loader every time it has status to
 * report.
 */
kern_return_t print_string(void *arg, printf_data_t string,
    unsigned int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our special prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

Handling a Status Message

You can receive many reply messages as the result of just one call to **kern_loader_status_port()**. The function you must use to handle these reply messages is defined as follows:

kern_return_t *string_func*(void **arg*, printf_data_t *string*, u_int *string_count*, int *level*)

The first argument, *arg*, has the same value as the **arg** field in the **kern_loader_reply_t** structure. The string that the kernel-server loader is logging is returned in *string*, with the string's length returned in *string_count*. The *level* argument is set to the priority of the log message, using the priorities defined in the header file **sys/syslog.h** (LOG_EMERG, LOG_ALERT, and so on).

Your function should return KERN_SUCCESS.

The following code is an example of a *string_func* named **print_string()**.

```
/*
 * This function is called by kern_loader every time it has status to
 * report.
 */
kern_return_t print_string(void *arg, printf_data_t string,
    u_int string_count, int level)
{
    /* If the string is empty, return. */
    if (string_count == 0 || !string)
        return KERN_SUCCESS;

    /* Print the string we were passed, with our special prefix. */
    printf("print_string: %s", string);

    return KERN_SUCCESS;
}
```

Handling a Synchronization Message

A call to **kern_loader_ping()** results in a single reply message. Your handler for this reply message must have the following syntax:

kern_return_t *ping_func*(void **arg*, int *id*)

The first argument, *arg*, is the value in the **arg** field of the **kern_loader_reply_t** structure. *id* is the same as the *id* value specified in the call to **kern_loader_ping()**. Your *ping_func* should return KERN_SUCCESS.

Here's an example of a *ping_func* that causes its task to shut down.

```
/* This function is called after a kern_loader_ping(). */
kern_return_t ping (void *arg, int id)
{
    exit(0);    /* Kill this process. */
}
```

Handling a Log Message

Each time you call **kern_loader_log_data()**, you receive a single reply message as soon as any log data from the specified driver is available. The function you write to handle this message must have the following syntax:

kern_return_t *log_func***(**void **arg*, printf_data_t *log_data*, unsigned€int€*log_data_count***)**

The first argument has the same value as the *arg* field in the **kern_loader_reply_t** structure. The *log_data* argument is a string containing the log entry from the loadable kernel server, preceded by a time stamp that indicates the relative time when the kernel-server loader received the log message. The *log_data_count* is the size of *log_data* in bytes. You should call **vm_deallocate()** on *log_data* when it's no longer needed.

Your *log_func* should return KERN_SUCCESS.

Here's an example of a *log_func* called **log_data**. It prints out the log message it's passed, and then requests another log message.

```
kern_return_t log_data(void *arg, printf_data_t log_data,
    unsigned int log_data_count)
{
    kern_return_t r;

    /* Print the string we were passed, with our prefix. */
    printf("log_data: %s\n", log_data);
    vm_deallocate(task_self(), (vm_address_t)log_data,
        log_data_count*sizeof(*log_data));

    /* Request the next log message. */
    r = kern_loader_get_log(kl_port, server_com_port, reply_port);
    if (r != KERN_SUCCESS) {
        kern_loader_error("log_data:  Error calling
            kern_loader_get_log", r);
        exit(1);
    }

    return KERN_SUCCESS;
}
```