Tru64 UNIX

Writing Kernel Modules

Part Number: AA-RHYGC-TE

September 2002

Product Version: Tru64 UNIX Version 5.1B or higher

This guide contains information systems engineers need to write kernel modules for the HP Tru64 UNIX operating system. It includes topics useful for device driver developers who can benefit from having an intermediate layer of code between the driver software and physical devices. It can also benefit third-party developers who want to augment the kernel with modules tailored to their particular environment.

Hewlett-Packard Company
Palo Alto, California

Contents

6 Symmetric Multiprocessing and Locking Methods

Tables

About This Manual

This manual discusses topics related to writing kernel modules for computer systems running the HP Tru64 UNIX operating system.

Audience

This manual is intended for systems engineers who:

- Understand the design and implementation of the Tru64 UNIX operating system and desire to enhance the functionality of the `/vmunix` kernel with kernel modules that they write
- Understand the basics of the CPU hardware architecture, including interrupts, direct memory access (DMA) operations, and I/O
- Use standard library routines to develop programs in the C language
- Know the Bourne or some other shell based on the UNIX operating system
- Understand basic UNIX operating system concepts, such as kernel, shell, process, configuration, and autoconfiguration
- Understand how to use the Tru64 UNIX programming tools, compilers, and debuggers
- Develop programs in an environment that involves dynamic memory allocation, linked list data structures, and multitasking

This manual assumes that you have a strong background in operating systems based on the UNIX operating system. It also assumes that you have a strong background in systems and C programming. In addition, the manual assumes that you have no source code licenses.

New and Changed Features

The following list summarizes changes and additions that have been made since the last release of this manual:

Information about kernel threads routines for RAD/NUMA systems has been added to the Kernel Threads chapter.

Throughout this book, by Tru64 UNIX documentation convention, the word “option” replaces “flag” (for example, compiler flag becomes compiler *option*).

Scope of the Manual

This manual is for users of the Tru64 UNIX operating system on computer systems developed by the Hewlett-Packard Corporation. It describes how to develop a kernel module and presents examples where kernel modules can be used. The manual also presents examples that show how to use routines that are for symmetric multiprocessing and kernel threads.

The manual assumes that you are new to writing kernel modules but may have experience writing device drivers or programming in the UNIX kernel.

Organization

The manual contains ten chapters and a glossary:

Chapter 1	Provides an overview of the information in this manual. Defines kernel modules, presents a high-level model for using kernel modules, presents reasons for writing a kernel module, and describes general rules for writing a kernel module.
Chapter 2	Describes how to initialize a kernel module using the `configure` routine.
Chapter 3	Describes setting module attributes and the module attribute table.
Chapter 4	Describes the boot timeline and how to implement callbacks in a kernel module.
Chapter 5	Describes programming capabilities available in kernel mode.
Chapter 6	Provides an overview of the SMP environment, including guidelines for selecting a locking method.
Chapter 7	Describes how to define and use simple locks in an SMP environment.
Chapter 8	Describes how to define and use complex locks in an SMP environment.
Chapter 9	Provides an introduction to multithreaded programming for kernel modules and discusses using kernel threads.
Chapter 10	Describes key steps for creating a single binary module (`.mod` file).
Glossary	

Related Documentation

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals

that meet their needs. (You can order the printed documentation from HP.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Writing kernel modules is a complex task; writers require knowledge in a variety of areas. One way to acquire this knowledge is to have at least the following categories of documentation available:

- Hardware documentation
- Bus-specific device driver documentation
- Operating system overview documentation
- Programming tools documentation
- Network programming documentation

The following sections list the documentation for each of these categories.

Hardware Documentation

If your kernel module is a device driver, have the hardware manual available for the device whose module you are coding. Also have access to the manual that describes the architecture for the CPU on which the driver operates, for example, the *Alpha Architecture Reference Manual*.

Bus-Specific Device Driver Documentation

Writing Device Drivers is the core manual for developing device driver kernel modules on the Tru64 UNIX Version 5.1B operating system. It contains information needed to develop modules on any bus that operates on HP platforms.

The Device Driver Reference Pages describe the routines, data structures, and global variables that device drivers use. These reference pages are available online in HTML format at the following location:

http://www.tru64unix.compaq.com/docs/dev_doc/DOCUMENTATION/HTML/DDK_R2/DOCS/HTML/REF_LIB.HTM

The following manuals provide information about writing device drivers for a specific bus:

- *Writing PCI Bus Device Drivers*

  This manual provides information for systems engineers who write device drivers for the PCI bus. The manual describes PCI bus-specific topics, including PCI bus architecture and data structures that PCI bus device drivers use.

- *Writing VMEbus Device Drivers*

  This manual contains information that systems engineers need to write device drivers that operate on the VMEbus. The manual describes VMEbus-specific topics, including VMEbus architecture and routines that VMEbus device drivers use.

Operating System Overview Documentation

See the *Technical Overview* for a technical introduction to the Tru64 UNIX operating system.

This manual provides a technical overview of the Tru64 UNIX system, focusing on the networking subsystem, the file system, virtual memory, and the development environment. This manual does not supersede the Software Product Description (SPD), which is the definitive description of the Tru64 UNIX system.

Programming Tools Documentation

To create your kernel modules, you use a number of programming development tools. Make sure that you have on hand the manuals that describe how to use these tools. The following manuals provide information related to programming tools that are used in the Tru64 UNIX operating system environment:

- *Kernel Debugging*

  This manual provides information about debugging kernels. The manual describes how to use use the `dbx`, `kdbx`, and `kdebug` debuggers to find problems in kernel code. It also describes how to write a `kdbx` utility extension and how to create and analyze a crash dump file. This manual is for system administrators who are responsible for modifying, rebuilding, and debugging the kernel configuration. It is also for system programmers who need to debug their kernel space programs.

- *Programming Support Tools*

This manual describes several commands and utilities in the Tru64 UNIX system, including facilities for text manipulation, macro and program generation, and source file management. The commands and utilities that this manual describes are primarily for programmers, but some of them (such as `grep, awk, sed,` and the Source Code Control System (SCCS)) are useful for other users. This manual assumes that you are a moderately experienced user of UNIX systems.

- *Programmer's Guide*

  This manual describes the programming environment of the Tru64 UNIX operating system, with an emphasis on the C programming language. This manual is for all programmers who use the Tru64 UNIX operating system to create or maintain programs in any supported language.

System Management Documentation

See the *System Administration* manual for information about building a kernel and for general information on system administration.

This manual describes how to configure, use, and maintain the Tru64 UNIX operating system. It includes information on general day-to-day activities and tasks, changing your system configuration, and locating and eliminating sources of trouble. This manual is for the system administrators who are responsible for managing the operating system. It assumes a knowledge of operating system concepts, commands, and configurations.

Reader's Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.

- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

This manual uses the following conventions:

. . .	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
file	Italic type indicates variable values, placeholders, and function argument names.
`buf`	In function definitions and syntax definitions used in module configuration, this typeface indicates names that you must type exactly as shown.
[ ]	In formal parameter declarations in function definitions and in structure declarations, brackets indicate arrays. Brackets also specify ranges for device minor numbers and device special files in file fragments. However, for syntax definitions, these brackets indicate items that are optional.
|	Vertical bars that separate items in syntax definitions indicate that you choose one item from among those listed.

1

Introduction to Kernel Modules

This chapter presents an overview of kernel modules by discussing the following topics:

- Definition of kernel module (Section 1.1)
- Purpose of a kernel module (Section 1.1.1)
- The kernel module environment (Section 1.1.2)
- Designing a kernel module (Section 1.1.3)
- Writing a kernel module (Section 1.2)

1.1 What Is a Kernel Module?

A **kernel module** is a binary image that contains code and data structures that runs in the UNIX kernel. It has the following characteristics:

- Is statically loaded as part of `/vmunix` or dynamically loaded into memory
- Runs in kernel mode
- Has a file name that ends with the extension `.mod`
- Contains a well-defined routine that executes first to initialize the module
- May be a **device driver** when it performs any one of these additional tasks:
  - Handles interrupts from hardware devices
  - Accepts I/O requests from applications

The kernel contains many modules, some of which are device drivers. In this manual, a kernel module is defined more broadly than a device driver because it can be used to perform a variety of functions, including:

- Management functions
- Common functions shared by other modules

1.1.1 Purpose of a Kernel Module

The kernel consists of a set of kernel modules that interact with each other, each of which performs a specific function. Some kernel modules perform software functions exclusively, while others (such as device drivers) control the operation of system hardware components.

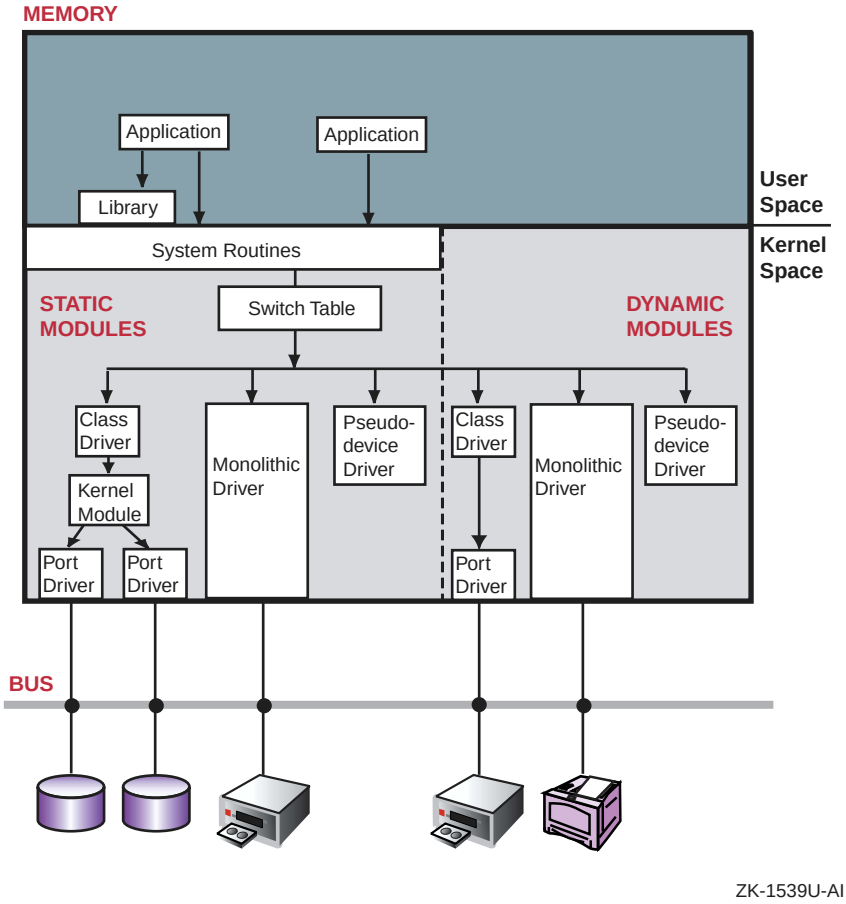
A purpose for writing a kernel module is to provide a middle layer of code, or common code, which increases the efficiency of your system by combining similar tasks in a single area and eliminating redundant code.

For example, assume that you need to write a SCSI driver for disk and tape peripheral devices. You can write two **monolithic drivers** — one for each hardware device — but this means replicating a majority of the code, while only a small amount differs. By writing a kernel that module contains common code, you eliminate this redundancy (see Figure 1–1). One class driver might handle SCSI tapes and another handle SCSI disks. Both call the kernel module, which sends the I/O request to a variety of port drivers. The port drivers send requests to the SCSI controller. As you add more disk or tape drives to your system, the kernel module seamlessly manages the expansion, while controller-specific code is confined to the new port drivers. Similarly, you can add a different SCSI device (for example, a scanner) by writing a new class driver. The kernel module maintains a consistent interface to the other kernel modules and make adding the new driver easier.

1.1.2 Kernel Module Environment

Figure 1–1 shows a kernel module in relationship to other modules in the kernel. As a binary image, a kernel module can be loaded statically as part of `/vmunix` or dynamically loaded into memory. In this example, the kernel module is part of a driver **subsystem**.

Figure 1–1: Kernel Module Environment



The following list describes the main components in Figure 1–1.

Application

A user-mode program that, in the context of this manual, makes various requests to the kernel modules. If a kernel module is part of a device driver, these requests typically perform I/O operations to hardware components. Another term for application is **utility**.

Bus

A hardware component that connects multiple buses and controllers to the system.

Class/Port Driver

The **class/port driver** comprises two drivers. The class driver supports user interfaces while the port driver supports the hardware and handles interrupts. The driver model is always made of more than one module and it can have multiple class drivers, multiple port drivers, and some common code in a middle layer. The structure of this driver eliminates code duplication.

Controller

A hardware component that performs a specific function, such as communicate on the network or control the graphics monitor.

Device

A hardware component that is connected to a controller.

Device Driver

A kernel module that supports one or more hardware components. There are two driver models: the monolithic driver model and the class/port driver model.

Interrupt

A signal from a hardware component that eventually causes the interrupt handler in the appropriate driver to be called.

Kernel Module

A `.mod` file residing in the kernel that executes common code. In Figure 1–1, the kernel module is part of a device driver.

Kernel Space

Activities that happen within the UNIX kernel. Modules may be statically loaded as part of `/vmunix` or dynamically loaded as needed. The **module framework**, which Figure 1–1 depicts as the background area of kernel space, loads modules, unloads modules, makes management requests, and keeps track of the modules in kernel space.

Library

User-mode code applications call. Libraries contain **routines** that perform common functions that many applications use.

Monolithic Driver

Kernel module code that is all-inclusive; supporting everything from user requests to processing interrupts from hardware.

Pseudodevice driver

A driver, such as the `pty` terminal driver, structured like other drivers but does not operate on a bus and does not control hardware. A pseudodevice driver does not register itself in the hardware topology (system configuration tree). Instead, it relies on the device driver method of the `cfgmgr` framework to create the associated device special files.

Switch Table

A data structure in the kernel where the block and character I/O interface entry points are stored.

System Routines

Routines in the kernel that can be called from user mode (applications and libraries).

User Space

User application level or command-line interface to the operating system.

1.1.3 Designing a Kernel Module

Consider the following guidelines when you design a kernel module:

- A kernel module is best written as a **single binary image** that can be statically loaded as part of `/vmunix` or dynamically loaded into memory.
- When you write your kernel module, it is important to design your code correctly with regard to **dispatch points** along the **boot timeline**. These are points along the boot path (timeline) that are reached as the operating system boots. When a dispatch point is reached, certain things are configured and made available. As a single binary image, your kernel module can be statically loaded as part of `/vmunix` or dynamically loaded into memory; therefore, any callbacks that you register must reflect the proper order of dispatch along the boot timeline (see Chapter 2).
- If you support dynamically loaded kernel modules, plan to write features that support dynamic unloading as well, for these reasons:
  - Unloading a module will free up resources.

– Dynamic unloading allows you to replace an old version of a kernel module with a new version without rebooting.

1.2 Writing a Kernel Module — Key Tasks

This section is organized so that key tasks for writing a kernel module are logically grouped:

- Section 1.2.1 describes required tasks that all kernel module writers need to perform to develop a kernel module.
- Section 1.2.2 describes optional tasks for writers whose modules run in an SMP environment or use kernel threads.
- Section 1.2.3 describes how to build a kernel module.

1.2.1 Required Tasks

All kernel module writers need to understand module initialization, creating the module attribute table, using callbacks, and working in kernel mode. The following sections describe these tasks.

1.2.1.1 Initializing a Kernel Module

Kernel module **initialization** occurs in both **static** and **dynamic** mode. Kernel module writers must understand the concept of a single binary image, the build-load-initialize sequence, and how to use the `configure` routine to perform initialization tasks to add a kernel module (make it known to the kernel) or to remove it. Chapter 2 describes these concepts and the required tasks for coding your kernel module to initialize properly. It also describes how to unload dynamically loaded modules.

1.2.1.2 Creating the Attribute Table

All kernel modules must contain an **attribute table**. Chapter 3 describes a variety of tasks you can perform on the module attribute table to retrieve data from the table and set data in the table.

1.2.1.3 Using Callbacks

Kernel modules contain one or more **callback routines** that perform different aspects of initialization along the boot timeline. Coding callback routines in a kernel module is a key task for creating a kernel module that may function as a single binary image. Chapter 4 describes the rules for using callbacks in a kernel module. It discusses callbacks in relation to dispatch points along the boot timeline, and how the kernel calls the kernel module's callback routine.

1.2.1.4 Working in Kernel Modules

You can perform many tasks in kernel mode. Chapter 5 describes how to:

- Work with string routines
- Use data copying routines
- Use kernel-related routines
- Work with system time
- Use kernel threads
- Use locks

1.2.2 Additional Tasks

If your kernel module executes in a **symmetric multiprocessing** (SMP) environment or uses kernel threads, you must perform additional tasks, as described in the following sections.

1.2.2.1 Working in an SMP Environment

Selecting a locking methodology and coding the correct type of lock in your kernel module are key tasks for writing kernel modules that execute in an SMP environment. Chapter 6 through Chapter 8 describe how to:

- Choose a locking methodology
- Use **simple lock** routines
- Use **complex lock** routines

1.2.2.2 Working with Kernel Threads

Chapter 9 describes the key concepts and tasks for developing kernel modules that use **kernel threads**. These include:

- Advantages of using kernel threads
- Kernel threads operations
- Kernel threads data structures
- Creating, starting, blocking, unblocking, and terminating thread processes

1.2.3 Building a Kernel Module

After you have written your kernel module, the next task is to build the executable module (a `.mod` file). Chapter 10 walks you through steps to build your kernel module.

2

Module Initialization

Kernel module **initialization** refers to the tasks necessary to incorporate a kernel module into the kernel and make it available for use by the system. After you write your kernel module, you create a single binary image (a file with the `.mod` extension) from the kernel module source file (usually a C file). This file is loaded into memory and its `configure` routine is called to perform initialization. Module initialization consists primarily of allocating and initializing data structures and calling on other kernel modules to inform them that your module is loaded and available.

The `configure` routine manages initialization. This chapter describes how this routine performs a variety of initialization tasks, including:

- Initializing the kernel module at system startup or at run time
- Preparing the kernel module for removal from the system

Other requests to the `configure` routine, such as reconfiguring the kernel module when an attribute value changes and returning information from the attribute table, are covered in Chapter 3.

2.1 The configure Routine

The `configure` routine handles requests that are targeted at the kernel module and performs the required actions. The `configure` routine's structure is the same for all kernel modules, regardless of the function that they perform and whether or not the kernel module is a device driver.

The naming convention for the `configure` routine requires that the name of the routine be the module name followed by `_configure`. This convention allows the module framework to locate the routine and call it. For example, for the kernel module `example.mod`, the `configure` routine is named `example_configure`.

If your module does not contain a properly named `configure` routine, one of the following conditions will occur:

- For statically loaded modules, the `/vmunix` kernel will not be able to build.
- For dynamically loaded modules, the module will not be able to load into memory.

2.1.1 Parameters

The `configure` routine accepts the following **parameters**:

`op (cfg_op_t)`

The module framework sets this parameter to one of several request codes that describe the operation for the module to perform:

- Initialize the module — `CFG_OP_CONFIGURE`
- Obtain attribute values — `CFG_OP_QUERY`
- Change attribute values — `CFG_OP_RECONFIGURE`
- Prepare the module for unloading — `CFG_OP_UNCONFIGURE`

These operations are described in Section 2.1.2.

`indata (cfg_attr_t *)`

Specifies a pointer to an array of data structures that contain information about the attributes in your kernel module attribute table, plus status information. The module framework determines the validity of attribute values when it copies attributes into memory, and it sets the status to indicate whether the value passes those tests.

`indatalen (size_t)`

Specifies the number of structures in the `indata` array.

`outdata (cattr_t)`

Specifies a pointer to a module-specific output data structure when the `op` parameter specifies a subsystem-defined operation. Otherwise, its value is `NULL`.

`outdatalen (size_t)`

Specifies the size of the `outdata` parameter in bytes.

Typically, the `configure` routine is written as a `switch` statement, with one `case` statement to handle each operation.

For example:

```
int example_configure (cfg_op_t op,
                       cfg_attr_t *indata,
                       size_t indatalen,
                       cattr_t *outdata,
                       size_t outdatalen)
{
    int status;.
    .
    .
    switch(op) {
      case CFG_OP_CONFIGURE:
        status=value;
```

```
        .
        .
        .
        break;
      case CFG_OP_QUERY:
        status=value;
        .
        .
        .
        break;
      case CFG_OP_UNCONFIGURE:
        status=value;
        .
        .
        .
        break;
      case CFG_OP_RECONFIGURE:
        status=value;
        .
        .
        .
        break;
      default:
      status=ENOTSUP;
        break;
      }

      return (status);
}
```

The ENOTSUP error return value indicates that the kernel module does not support the requested operation. Otherwise, the routine returns a status value appropriate for the request. (See Section 2.1.3 for information on return status values.)

2.1.2 Request Codes

The `configure` routine accepts several parameters. (See Section 2.1.1 for a list of all parameters accepted by the `configure` routine.) The `op` parameter takes one of the following request codes, which describe the specific operation for the module to perform:

- `CFG_OP_CONFIGURE`

  When the module framework calls the `configure` routine with the `CFG_OP_CONFIGURE` request code, the kernel module begins initialization. In this way, the `configure` routine functions similarly to the `main()` routine in a user program. Your kernel module must be initialized whether it is loaded dynamically or statically. Section 2.2.1 describes this operation in more detail.

- `CGF_OP_QUERY`

  This request code retrieves values of attributes defined in the module attribute table. The kernel module initializes the values of attributes that are stored in the module attribute table so that the proper values are retrieved. (See Chapter 3 for more information.)

- CFG_OP_RECONFIGURE

  This request code specifies that values for some attributes in the module attribute table have been set and that the kernel module operates based on changes to the values of the attributes. (See Chapter 3 for more information.)

- CFG_OP_UNCONFIGURE

  This request code specifies that an attempt to unload your module has been requested, which results in either module cleanup or a return error. In effect, this request code asks that your module undo the initialization tasks that CFG_OP_CONFIGURE performed and prepare it for removal from the system. (See Section 2.2.2 for more information.)

2.1.3 Return Status Values

The configure routine may return any standard status value from the file /usr/include/errno.h as an int to the module framework. The following list defines the most common return status values:

- ESUCCESS — Indicates success.
- ENOMEM — Indicates insufficient memory.
- ENOTSUP — Indicates that the operation is not supported.
- ENOSYS — Indicates that the operation is not supported at this time. It may have been called too early and is supported later in the boot timeline.
- EINVAL – Indicates that an unrecognized parameter was passed (for example, indata, indatalen).

The return status value is later appended to the higher 16 bits of a final return that is returned to the caller. The module framework status resides in the lower 16 bits of the return status.

2.2 Module Initialization

Before a kernel module can be useful, it typically needs to initialize data structures and inform other kernel modules that it exists and is available. The module framework calls the configure routine with the CFG_OP_CONFIGURE request code to alert the module to perform initialization. Likewise, the module framework passes the CFG_OP_UNCONFIGURE request code to alert the kernel module to prepare for removal from the system. These codes are described in detail in the following sections.

2.2.1 Receiving the CFG_OP_CONFIGURE Request

The module framework calls the `configure` routine with the `CFG_OP_CONFIGURE` request code to request that the module perform its one-time initialization. This is always the first call into the module, regardless of whether it is statically or dynamically loaded. If the kernel module is statically loaded, the module framework calls the `configure` routine very early in the boot timeline. Because of this, the kernel module typically registers callback routines to execute immediately or at specific dispatch points to perform initialization tasks. These tasks include:

- Allocating data structures
- Initializing locks
- Starting kernel threads
- Registering with other subsystems

When you code your kernel module initialization process using callbacks, the result is a single binary image that can be loaded statically or dynamically. Otherwise, your kernel module will be either a static module or a dynamic module, but not both. Chapter 4 expands on this concept by discussing the relationship between callbacks and dispatch points. The following sections present further considerations for modules that are loaded either statically or dynamically.

2.2.1.1 Implementing Statically or Dynamically Loaded Kernel Modules

When a kernel module is statically loaded, it is linked as part of `/vmunix` and loaded into memory as part of the kernel. The module framework must call the `configure` routine with the `CFG_OP_CONFIGURE` request code before memory can be allocated, locks can be used, and subsystems can be used. As a result, a statically loaded module typically is not able to perform initialization when its `configure` routine is called with the `CFG_OP_CONFIGURE` request code. Instead, it registers callbacks that are invoked when these resources become available as the system boots. In contrast, a dynamically loaded module is linked as its own image and loaded into memory on its own. If you used callbacks in a dynamically loaded module, the initialization still occurs properly.

To overcome the problem of resources not being available for a statically loaded module, the `configure` routine registers callback routines to be called at specific dispatch points, as described in Chapter 4. Initialization takes place when these callback routines are called. Callbacks enable your module to be a single binary image that can be statically or dynamically loaded.

2.2.1.2 Tracking the Configuration

To handle initialization correctly, whether your module is statically loaded or dynamically loaded, global variables keep track of the following information:

- Whether the kernel module has already been initialized

  A kernel module receives the CFG_OP_CONFIGURE request code only once. Therefore, you define a global variable to keep track of this information and set the variable's initial value to FALSE. When the configure routine successfully accepts the CFG_OP_CONFIGURE request, set this value to TRUE. For example, for a kernel module that is named example.mod, the module defines the example_config global variable as follows to indicate whether the module has been initialized:

  ```
  int example_init_config = FALSE;
  ```

- Whether the module was dynamically or statically loaded

  The module framework returns the current configuration state when you call the cfgmgr_get_state routine. The cfgmgr_get_state routine returns SUBSYSTEM_STATICALLY_CONFIGURED if the module was statically loaded. It returns SUBSYSTEM_DYNAMICALLY_CONFIGURED if the module was dynamically loaded. Your module can call this routine if it needs to know how it was loaded. Typically, you write a kernel module so that it does not need to call this routine.

- Whether the module's callback routines completed successfully

  When the kernel module is configured at startup, callback routines run at different times along the boot timeline. Therefore, global variables are the only way to communicate the success or failure of the callback routines. For example, you do not want to perform any postconfiguration operations if the preconfiguration callback routine failed.

  In this example, the following global variable is defined to hold the callback status:

  ```
  int example_inited=EFAIL;
  ```

  The global variable is defined with an error status. When the kernel module is loaded, the callback routine has not yet been called. The callback routine stores its status in this global variable before it returns to the caller. This status is available to the remainder of the source code in the module for the purpose of determining the status of the callback routine (that is, it determines whether the module has been successfully initialized).

2.2.1.3 Allocating Memory for Data Structures

Your kernel module may need to allocate memory for data structures during initialization. You must wait until the CFG_PT_VM_AVAIL dispatch point occurs. When you are ready to allocate memory, use the MALLOC macro.

(Use the FREE macro to deallocate memory.) See Section 5.3.7 for more information about allocating memory.

2.2.2 Receiving the CFG_OP_UNCONFIGURE Request

The module framework calls the configure routine with the CFG_OP_UNCONFIGURE request code to have both statically loaded and dynamically loaded kernel modules prepare to go off line. When modules are brought off line, they are not available for use by any other module in the kernel. Only dynamically loaded kernel modules can actually be unloaded. Statically loaded modules remain loaded after they are brought off line. (See Section 2.2.1.2 to determine how the kernel module was loaded.)

When a module (static or dynamic) has successfully gone off line, it returns ESUCCESS.

To prepare to go off line, the kernel module must accomplish the following tasks before returning a success status value to the module framework:

- Deallocate all data structures
- Deinitialize locks
- Terminate all kernel threads
- Unregister with other kernel subsystems

A kernel module (static or dynamic) can determine that it cannot be unloaded. In this case, the module returns an error to the module framework to keep it from attempting to unload the module.

3

Module Attributes

This chapter describes the module attribute table and the operations that can be performed on it to:

- Retrieve data from the table
- Set data in the table

It also describes entries in the table and how to manipulate the values of the attributes.

3.1 The Attribute Table

Every kernel module must have one attribute table that defines some of the data for the kernel module. The system administrator can use settable attributes in the attribute table to tune the module.

Note

If your kernel module does not have any defined attributes, you must still provide an attribute table with one terminating NULL entry.

The name of the attribute table is the module name followed by `_attributes`. For example, for the `example.mod` kernel module, the attribute table is named `example_attributes`.

The attribute table is an array of the data structure `cfg_subsys_attr_t` (defined in `/usr/include/sys/sysconfig.h`). Each `cfg_subsys_attr_t` data structure defines an attribute for your module. There are no required attributes for this table.

An **attribute table entry** comprises one instance of the `cfg_subsys_attr_t` data structure. The last table entry must be all zeros. Section 3.2 describes the fields in an attribute table entry.

3.2 Attribute Table Entry

An attribute table entry is one instance of the `cfg_subsys_attr_t` data structure. An entry is composed of many fields, which are defined in the following list:

- `addr (caddr_t)`

  Specifies the kernel address of the location that holds the value of the attribute. Using this address, the module framework returns the attribute's value during a `CFG_OP_QUERY` request and changes the value with a `CFG_OP_RECONFIGURE` request. As a result, the `configure` routine does not need to do additional processing. If you do not provide an address, the `configure` routine must separately handle value retrieval and deposit.

  If the attribute supports the `CFG_OP_CONFIGURE` or `CFG_OP_RECONFIGURE` request operation, then the address given in this field must be a writable location. That is, do not make it a location of the type `const`.

- `name (char [])`

  Specifies the ASCII name of the attribute. The name must be between two and `CFG_ATTR_NAME_SZ` characters in length, including the terminating null character.

  To create a name for your attribute, follow these conventions:

  – Use lowercase letters, unless capitals make better sense (for example, when using an acronym in the attribute name, such as `MAC_address`).

  – Use an underscore to separate parts of the name.

  – Create intuitive names; do not overabbreviate names.

  – Do not begin the name of the attribute with either `Method_` or `Device_`. The module framework reserves names that begin with these **strings**.

- `min_val` and `max_val (ulong)`

  Define the mininum and maximum allowed values for the attributes. The module framework interprets the contents of these two fields differently, depending on the data type of the attribute. If the attribute is one of the integer data types, these fields contain the minimum and maximum integer values the attribute can have. For attributes with the `CFG_ATTR_STRTYPE` data type, these fields contain the minimum and maximum lengths of the string. For attributes with the `CFG_ATTR_BINTYPE` data type, these fields contain the minimum and maximum numbers of bytes allowed.

- `val_size (ulong)`

  If the attribute is a binary type, this field contains the current size (in bytes) of the attribute value. This field is not used if the attribute is an integer or string.

- `type (uchar)`

  Specifies the data type of the value for this attribute. See Section 3.2.1 for a list of values for this field.

- `operation (uchar)`

  Specifies the operations that the module allows on this attribute (for example, initialize or query). This field is a bit mask. See Section 3.2.2 for a list of values for this field.

3.2.1 Attribute Data Types

The following data types are supported for attribute table entries:

`CFG_ATTR_STRTYPE` — A null-terminated array of characters
`CFG_ATTR_INTTYPE` — A 32-bit signed integer
`CFG_ATTR_UINTTYPE` — A 32-bit unsigned integer
`CFG_ATTR_LONGTYPE` — A 64-bit signed integer
`CFG_ATTR_ULONGTYPE` — A 64-bit unsigned integer
`CFG_ATTR_BINTYPE` — An array of bytes
`CFG_ATTR_UCHARTYPE` — An 8-bit unsigned char (byte)
`CFG_ATTR_USHORTTYPE` — A 16-bit unsigned short
`CFG_ATTR_INTARRAYTYPE` — An array of 32-bit signed integers
`CFG_ATTR_UINTARRAYTYPE` — An array of 32-bit unsigned integers
`CFG_ATTR_LONGARRAYTYPE` — An array of 64-bit signed longs
`CFG_ATTR_ULONGARRAYTYPE` — An array of 64-bit unsigned longs
`CFG_ATTR_STRARRAYTYPE` — An array of pointers to null-terminated strings

3.2.2 Operations Allowed on an Attribute

You can set the `operation` field in an attribute table entry to any combination of the following request codes:

`CFG_OP_CONFIGURE`

The value of the attribute can be modified during initialization using a data value from the `/etc/sysconfigtab` file. (Section 10.1.5 describes how to create the `/etc/sysconfigtab` file.) If the kernel address for the attribute is specified in the attribute table, the initialization occurs before the module framework calls the kernel module's `configure` routine with the `CFG_OP_CONFIGURE` request

code. If the attribute's address is not specified, the `configure` routine must perform the modification itself.

Setting this option in the operator field allows the system administrator to set the value of the attribute through the `/etc/sysconfigtab` file. This gives the system administrator the ability to tune your module each time that it is loaded.

`CFG_OP_QUERY`

Setting this option allows users or applications to retrieve the value of the attribute. The module framework can read the attribute and return it to applications. The attribute's value is retrieved after the module framework calls the kernel module's `configure` routine with the `CFG_OP_QUERY` request code. (See Section 3.3.)

`CFG_OP_RECONFIGURE`

Setting this option allows users or applications to modify the value of the attribute at any time after the kernel module is up and running. The module framework sets the value before it calls the `configure` routine with the `CFG_OP_RECONFIGURE` request code. (See Section 3.4.)

`CFG_HIDDEN_ATTR`

Setting this option prevents the attribute from being displayed in the output of a `cfg_subsys_query_all` operation.

Note

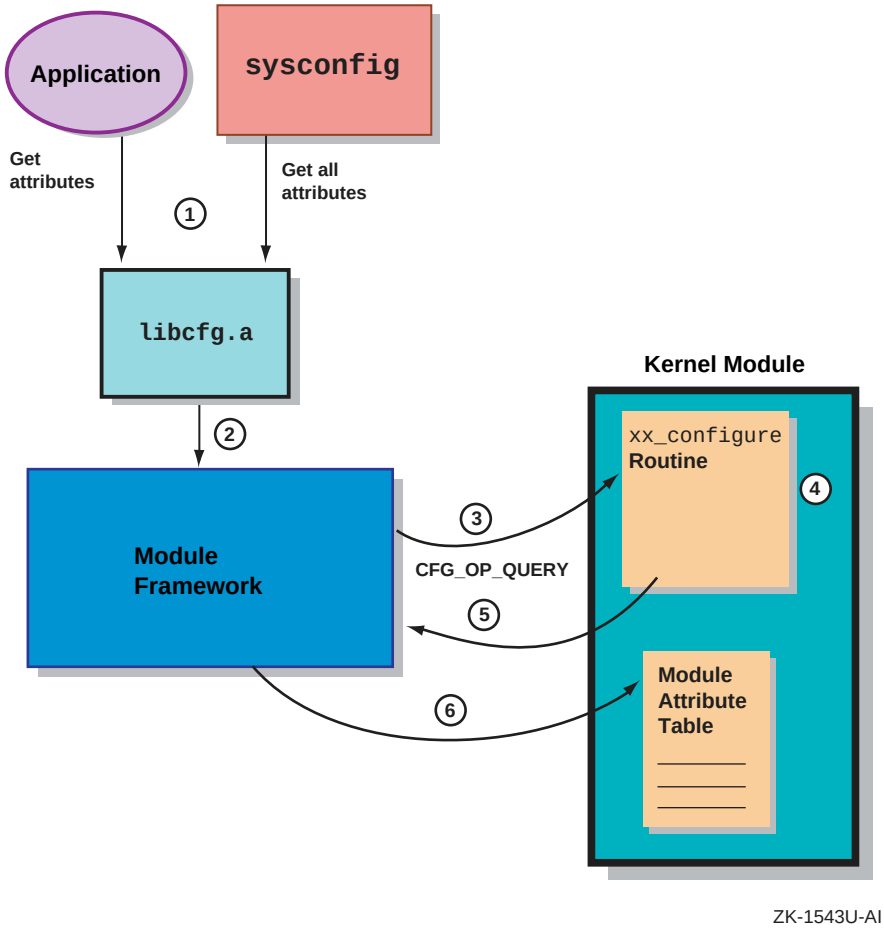
If you do not specify the kernel address of an attribute in the attribute table, the `configure` routine must handle the setting, resetting, and retrieval of the attribute value by itself. The module framework cannot perform these actions automatically unless you supply the kernel address of the attribute.

3.3 Attribute Get Requests

When an application wants to get attribute values of a kernel module, it calls the `cfg_subsys_query(3)`routine or the `cfg_subsys_query_all(3)` routine in the `/usr/ccs/lib/libcfg.a` library. The library makes the request to the module framework.

The module framework validates the requests to get the valid attribute values. After successful validation, the module framework calls the `configure` routine with the `CFG_OP_QUERY` request code. Figure 3–1 shows these relationships.

Figure 3–1: Attribute Get Requests



The following list presents the sequence of steps in an attribute get request:

1. The application requests specific attributes or all attributes by calling the appropriate library routine in `/usr/ccs/lib/libcfg.a`.
2. The library passes the request to the module framework.
3. The module framework calls the `configure` routine with `CFG_OP_QUERY`.
4. The `configure` routine handles returning values for the requested attributes whose address is not specified in the attribute table.
5. The `configure` routine returns control to the module framework.
6. The module framework handles returning values for the requested attributes whose address is specified in the attribute table.

Consider the following when you use attribute get requests:

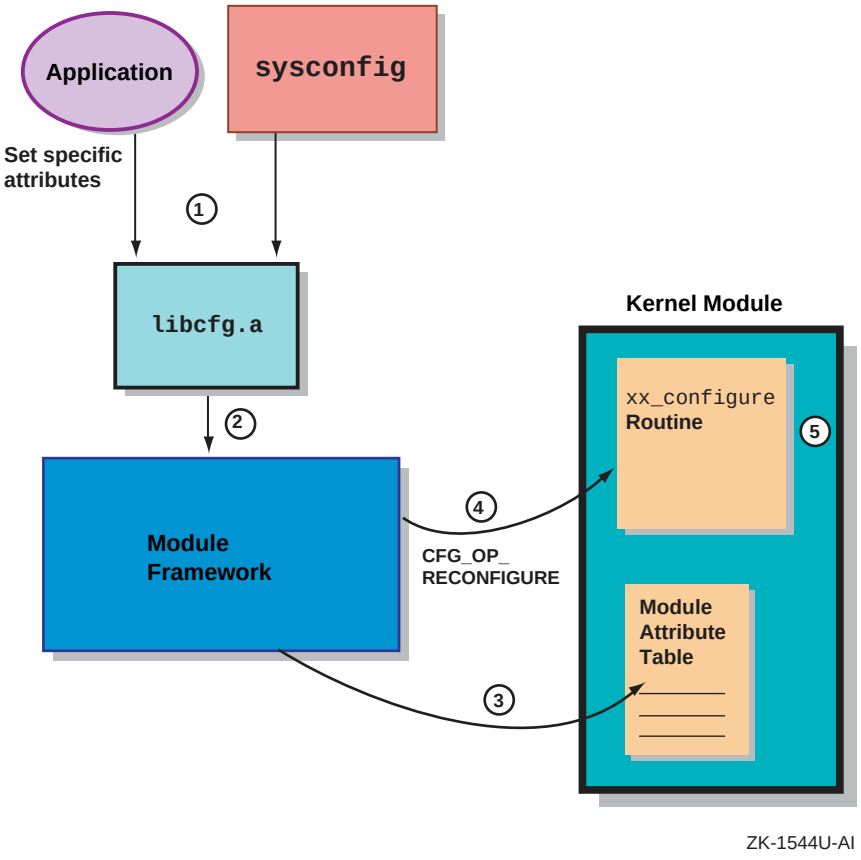
- You do not have to process a CFG_OP_QUERY request in your configure routine; you can simply return ESUCCESS.
- If you do not keep some or all of your attributes up to date, you may bring them up to date when you receive the CFG_OP_QUERY request.
- To determine which attributes are being requested, use the indata parameter. The status field of each cfg_attr_t data structure whose address is available to the module framework will be set to CFG_ATTR_PENDING during the CFG_OP_QUERY request; whereas the status field for each attribute that cannot be handled by the module framework will be set to CFG_ATTR_ESUBSYS.

3.4 Attribute Set Requests

When an application wants to set attribute values of a kernel module, it calls the cfg_subsys_reconfig routine in the /usr/ccs/lib/libcfg.a library. The library makes the request to the module framework.

The module framework sets the values of the requested attributes, then calls the configure routine with the CFG_OP_RECONFIGURE request code. The kernel module evaluates these values and functions accordingly. Figure 3–2 shows these relationships.

Figure 3–2: Attribute Set Requests



The following list presents the sequence of steps in an attribute set request:

1. The application requests to set the values of specific attributes.
2. The library passes the request to the module framework.
3. The module framework determines whether the new value falls within the range that is specified in the attribute table, and sets the status of each attribute. If the value determination succeeds, the module framework sets the attribute's value to the new value.
4. The module framework calls the `configure` routine with `CFG_OP_RECONFIGURE`.
5. The kernel module evaluates the new values and executes based on those values.

4

Dispatch Point Callbacks

This chapter describes callbacks in relation to dispatch points along the boot timeline and the rules for implementing them in your kernel module. Kernel modules may contain one or more callback routines, which perform different tasks at different dispatch points. The kernel interacts with the callback routines to perform these tasks at the appropriate time.

This chapter contains the following information:

- Understanding the UNIX boot timeline and how callbacks are affected (Section 4.1)
- Why to use callbacks in your kernel module (Section 4.2)
- Understanding dispatch points along the UNIX boot timeline (Section 4.3)
- How to implement callbacks in your kernel module (Section 4.4)

4.1 Understanding the UNIX Boot Timeline

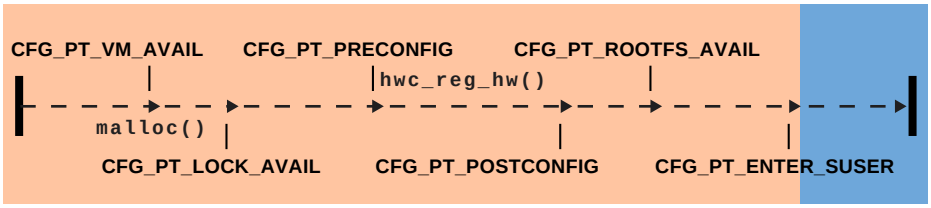
To understand why callbacks are needed and how to implement them, you need to understand some details of the UNIX boot timeline.

The boot timeline represents all code that executes while the system boots. Key to the boot process are dispatch points indicating certain functions can be done. While the system is booting, dispatch points occur in a specifically ordered manner (see Section 4.3). For example, the kernel-mode dispatch point `CFG_PT_VM_AVAIL` indicates the point where virtual memory can be allocated. Any activity that your module performs that requires the allocation of virtual memory must happen at or after this dispatch point. After single-user mode is reached, the dispatch points are more loosely ordered.

Callbacks are the mechanism for ensuring that the code in your module executes at the right point along the boot timeline. Section 4.4 describes ways that you can code your callback routine and, consequently, register the callback in your kernel module.

Figure 4–1 shows the boot timeline and the associated dispatch points.

Figure 4–1: Dispatch Points Along the Boot Timeline



ZK-1542U-AI

The arrows along the timeline depict the dispatch points. The routines that are shown in the example can be called at any time at or after the dispatch point is reached, but not before.

4.2 Why Use Callbacks?

Many kernel modules are dynamic modules — that is, they are dynamically loaded into memory as needed. Other kernel modules are statically loaded as part of `/vmunix` early in the boot timeline. For a kernel module to be a single binary image, it must be able to load statically as part of `/vmunix` or load dynamically as needed.

As explained in Chapter 2, when a module is loaded into memory, the only routine in the module that is known to the operating system is the `configure` routine. The module framework has access to the `configure` routine because of the predetermined name of the routine. (The module framework knows to look for a routine name that ends with `_configure`). The framework calls the `configure` routine at initialization so that the kernel module can register its other routines with the rest of the operating system.

When static kernel modules are called to initialize themselves, they cannot allocate memory, initialize locks, or call any routine that is not yet available on the boot timeline. For example, the call to initialize a kernel module (well before `CFG_PT_VM_AVAIL`) occurs early in the boot timeline, while the dispatch point for locking (`CFG_PT_LOCK_AVAIL`) occurs later (Figure 4–1). To avoid the problem of calling routines that are not yet available, the kernel module can register a callback routine that will be called later in the boot timeline. When that routine is called, it will perform the required initialization correctly because the routines it requires will be available.

Callbacks, then, are the mechanism for implementing kernel modules as single binary images. Statically loaded kernel modules register callbacks that the module framework can execute at a later time. For a static configuration, callbacks are registered to execute at dispatch points along the boot timeline.

For example, the device switch subsystem is statically configured. It registers a callback routine to initialize the in-memory copy of the database after virtual memory is available (at the dispatch point called CFG_PT_VM_AVAIL). It registers another callback routine to update the on-disk database files, if necessary. This callback occurs after the root file system becomes writable (at dispatch point CFG_PT_ROOTFS_WR) because the subsystem's files reside on the root file system.

For a dynamically loaded module, callback routines that register with the dispatch points along the boot timeline are called directly from the register_callback routine because the dispatch point has already occurred. This behavior is particular to dispatch points up to and including CFG_PT_ENTER_SUSER. All higher numbered dispatch points represent run-time (as opposed to boot-time) events. Their associated callback routines are invoked only whent the event reoccurs (or reoccurs).

Kernel modules call the register_callback routine to register their own callback routine. The kernel calls this routine when the specified dispatch point occurs.

4.3 Dispatch Points on the Boot Timeline

This section presents a list of dispatch points as they occur on the boot timeline. During system boot-up (prior to single-user mode), the dispatch points occur in a strict chronological order.

CFG_PT_HAL_INIT
Description: Hardware architecture layer is initialized.

CFG_PT_UNIXTBL_AVAIL
Description: Dynamically sized tables have been allocated.

CFG_PT_VM_AVAIL
Description: Virtual memory is available.
Common routines available: Device switch routines.

CFG_PT_LOCK_AVAIL
Description: Locking is available.
Common routines available: Routines that handle hardware registration.

CFG_PT_PRECONFIG
Description: **Scan** for hardware and preconfigure.

CFG_PT_TOPOLOGY_CONF
Description: The topology configuration point. The operating system can create threads, timeouts begin working, kernel event management is available, and the system begins incrementing time.

CFG_PT_PLATFORM_CONF
Description: The platform-specific configuration is about to occur.

CFG_PT_POSTCONFIG
Description: Postscan the hardware. Tasks that require completion of hardware configuration can be performed at this dispatch point. Hardware events are posted.

CFG_PT_CLU_CONF
Description: Cluster subsystem configuration may now be performed.

CFG_PT_GLROOTFS_AVAIL
Description: Global root file system has been mounted.

CFG_PT_ROOTFS_AVAIL
Description: Root file system has been mounted read-only. Tasks that require completion of the root file system mount operation can be performed at this dispatch point. Dynamic device registration can occur.

CFG_PT_ENTER_SUSER
Description: Enter single-user mode.

CFG_PT_ROOTFS_WR
Description: The root file system has become writable.

4.4 Implementing Callbacks in Your Kernel Module

This section describes how you code callbacks in your kernel module.

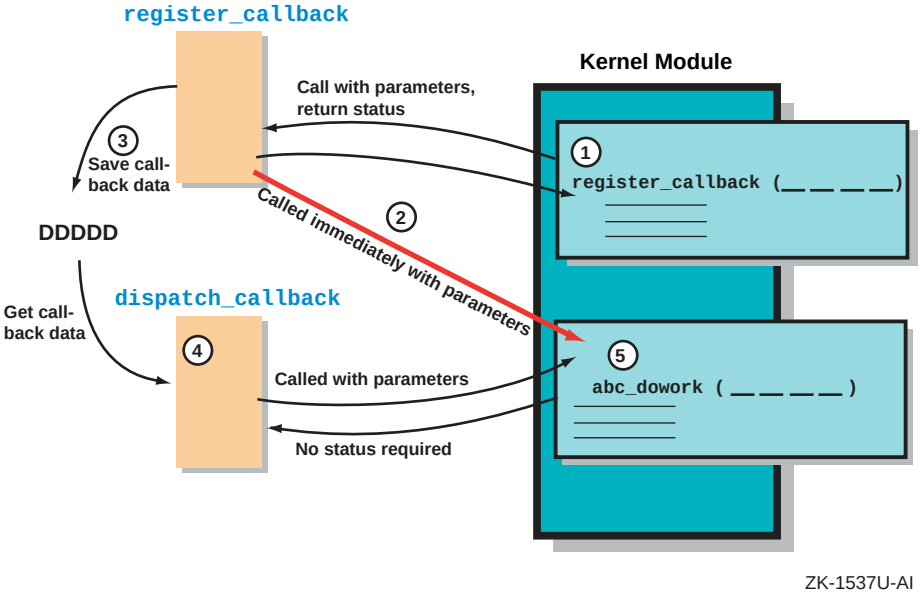
4.4.1 Coding Callbacks

To implement callbacks in your kernel module, you must:

- Call the `register_callback` routine
- Write a callback routine in your kernel module that will be passed parameters from the kernel's callback subsystem

Section 4.4.1.1 describes the first step in this process, registering your callback routine. It defines the parameters that are passed to the callback subsystem when you register callbacks. Section 4.4.1.2 describes how to write a callback routine in your kernel module that receives information from the callback subsystem prior to performing some task. Figure 4–2 shows how the kernel module uses the kernel's callback subsystem.

Figure 4–2: Using the Kernel Callback Subsystem



1 Some routine, typically the `configure` routine, calls `register_callback` because it needs the kernel module callback routine (`abc_dowork` in the example) called at some later point. When you call `register_callback` to register your callback routine, you pass several parameters: the dispatch point, the priority, the address of the callback routine, and an **argument** to be passed to the callback routine.

When `register_callback` is called, it does either step 2 or step 3:

2 The `register_callback` routine calls `abc_dowork` directly if the kernel dispatch point is on the boot timeline and it has already occurred. This completes the callback sequence.

3 The `register_callback` routine saves information about the callback and proceeds to the next step in the callback sequence. (This is the normal operation.)

4 The routine `dispatch_callback` calls the kernel module callback routine `abc_dowork` at the appropriate dispatch point.

5 The kernel module callback routine executes.

4.4.1.1 Calling the register_callback Routine

The `register_callback` routine enables your kernel module to execute its callback routine by storing callback information until the correct dispatch point. The `register_callback` routine has the following format:

```
int register_callback(void  (*func)(), int point, int order, ulong arg);
```

where

- The `func` parameter is the name of the callback routine that you want called at a particular dispatch point.
- The `point` parameter is the value of the dispatch point at which you want your callback routine called (for example, `CFG_PT_VM_AVAIL`).
- The `order` parameter is used to order multiple callback requests that are registered for the same dispatch point. A request with a smaller order value is executed before a request with a larger value. A kernel module may use this to coordinate among other modules. The order constant that is most useful to kernel module writers is `CFG_ORD_DONTCARE`. This constant registers the callback with no specific order priority.

  If you are a device driver writer, consider using one of the following order constants:

  `CFG_ORD_NOMINAL` — Registers the callback with lowest order priority.
  `CFG_ORD_MAXIMUM` — Registers the callback with the highest order priority.

- The `arg` parameter is used by the kernel module to communicate information to the callback routine. Pass the integer `0L` to indicate that you do not want to pass an argument.

When you call `register_callback` to register your callback routine, the information you pass says, in effect, "At this dispatch point, with this priority, call the kernel module callback routine with this argument." Normally, the callback will occur later than the `register_callback` call. There is one exception: if the callback being registered is for a dispatch point along the boot timeline that has already passed, the callback occurs immediately.

Upon successful completion, the `register_callback` routine returns the status value `ESUCCESS`. Otherwise, it returns one of the following error status values:

`ENOMEM` — The system limit on the maximum number of registered callbacks was exceeded. You can correct this error by increasing the value of the `max_callbacks` attribute in the `cm` subsystem and then rebooting the system. (See *System Configuration and Tuning* for details.)
`EINVAL` — The value that you passed as the `point` argument is outside of the minimum and maximum range.

A kernel module calls the `unregister_callback` routine to undo the registration of a callback. It has the following format:

```
int unregister_callback(void  (*func)(), int point, int order, ulong arg);
```

where the parameters are identical to those used by `register_callback`. Some callbacks may never be unregistered.

4.4.1.2 Writing the Callback Routine

When a callback occurs, the kernel executes the callback routine that you specified in the call to `register_callback`. The callback routine does all the callback processing and implements whatever action that you require when the callback occurs. The callback routine is most often written as part of your kernel module. It can be statically linked to the kernel as part of `/vmunix` or dynamically loaded at run time. The requirement is that it exists in the kernel prior to when the callback occurs.

The callback routine that you write in your kernel module is passed the dispatch point, order, and argument parameters when it is called.

A kernel module callback routine must conform to the following format:

```
void xx_callback(int point, int order, ulong arg, ulong arg2);
```

where the parameters are defined as follows:

- The `point` parameter is the value for the dispatch point. The value from the same parameter in the corresponding call to `register_callback` is passed.
- The `order` parameter specifies the order in which the callback routine is being called. The value from the same parameter in the corresponding call to `register_callback` is passed.
- The `arg` parameter specifies the argument that the kernel module requested to pass to the callback routine. The value from the same parameter in the corresponding call to `register_callback` is passed.
- The `arg2` parameter is an additional value supplied by the callback dispatcher. It is used to communicate point-specific information to the callback routine. For many dispatch points, this parameter is not used.

4.4.2 Registering Callbacks

To code callbacks in your kernel module, register all the callbacks in your `configure` routine. The following pseudocode fragment for `abc_configure.mod` registers two callbacks from within the `configure` routine:

```
.
.
abc_configure (opcode, ...){
   switch (opcode) {
   case CFG_OP_CONFIGURE:
      register_callback (abc_vm, CFG_PT_VM_AVAIL, CFG_ORD_DONTCARE, arg1)
    .
      register_callback (abc_post, CFG_PT_POST_CONFIG, CFG_ORD_DONTCARE, arg2)
    .
    .
    .
    }
}
abc_vm (int point, int priority, int arg){
.
.

}
abc_post (int point, int priority, int arg){
.
.

}
```

Note

Because there are a limited number of callbacks that you can use, we recommend that you do not register a large number of callback entries.

4.4.3 Nesting Callbacks and Unregistering Callbacks

A kernel module can register multiple callbacks, possibly at different callback points, by calling `register_callback()` many times. Callbacks may not, however, be nested — calling `register_callback()` from within a callback routine is illegal.

To enable deregistration, call `unregister_callback()` from within a callback routine. Doing so allows a callback to unregister itself or other callbacks.

4.4.4 Defining New Dispatch Points in Your Kernel Module

You can write a kernel module that uses the predefined dispatch points (see Section 4.3), or you can write a module that defines and uses new ones. The following steps describe how to define a new kernel dispatch point:

1. Choose and reserve a unique number for the new dispatch point.

   The valid range for developer-defined dispatch points is listed in the `/usr/include/sys/sysconfig.h` file, along with the values for the system-defined dispatch points.

Values for developer-defined run-time dispatch points that are triggered within the kernel must be within the range of these values: CFG_PT_RUNTIME_KERN_MIN_EXT (20000) to CFG_PT_RUNTIME_KERN_MAX_EXT (29999).

Values for developer-defined run-time dispatch points that are triggered outside the kernel (user mode) must be within the following range: CFG_PT_RUNTIME_USER_MIN_EXT (30000) to CFG_PT_RUNTIME_USER_MAX_EXT (39999).

2. Trigger the callback.

   All kernel callbacks triggered within the kernel are activated by the dispatch_callback() routine, which has the following format:

```
dispatch_callback (CFG_PT_MYPOINT, arg2)
```

   where CFG_PT_MYPOINT is the unique value for the dispatch point you define and arg2 communicates point-specific information to the callback routine. Therefore, when you define a dispatch point that is triggered from the kernel, you need to insert the dispatch_callback() call at the appropriate place within your kernel module.

   In contrast, when you define a dispatch point triggered from user space, you do not need to supply the dispatch_callback() call in the kernel module. A callback that is triggered from user mode is accomplished by setting the value of the user_cfg_pt attribute in the generic subsystem to the value of the dispatch point. For example, if you define a dispatch point that is triggered in user mode with a value of 35600, the following command triggers callbacks that are registered for this dispatch point:

```
sysconfig -r generic user_cfg_pt=35600
```

   To trigger the callback, you execute the command from within a script or from the user prompt. Alternately, you could call the cfg_subsys_reconfig routine from within a program to achieve the same result.

5

Kernel-Mode Capabilities

Tru64 UNIX offers several kernel-mode programming capabilities. This chapter describes the tasks that you can do in kernel mode:

- Work with string routines (Section 5.1)
- Use data copying routines (Section 5.2)
- Use kernel-related routines (Section 5.3)
- Manage system time (Section 5.4)
- Use kernel threads (Section 5.5)
- Use locks (Section 5.6)

This chapter discusses the routines most commonly used and provides code fragments to show how to call them in a kernel module. These code fragments and associated descriptions supplement the reference page descriptions for these routines.

5.1 Using String Routines

String routines allow kernel modules to:

- Compare two null-terminated strings (Section 5.1.1)
- Compare two strings by using a specified number of characters (Section 5.1.2)
- Copy a null-terminated character string (Section 5.1.3)
- Copy a null-terminated character string with a specified limit (Section 5.1.4)
- Return the number of characters in a null-terminated string (Section 5.1.5)

The following sections describe the routines that perform these tasks.

5.1.1 Comparing Two Null-Terminated Strings

To compare two null-terminated character strings, call the `strcmp` routine. The following code fragment shows a call to `strcmp`:

```
.
.
register struct device *device;
```

```
struct controller *ctlr;
.
.
.
if (strcmp(device->ctlr_name, ctlr->ctlr_name)) {[1]
.
.
.
}
```

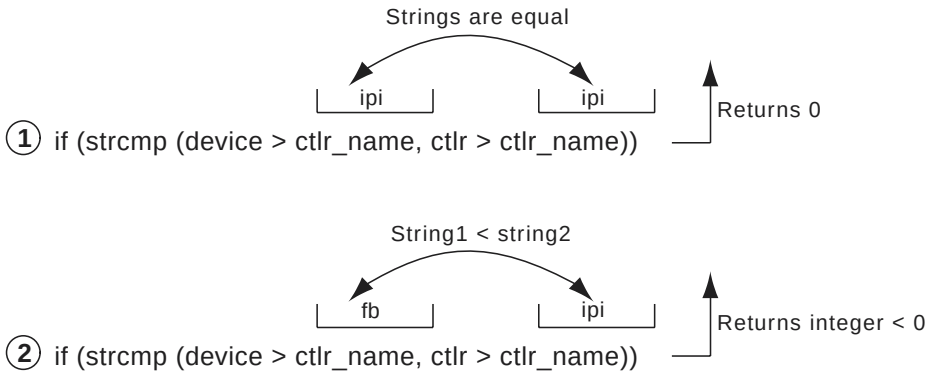
[1] Shows that the `strcmp` routine takes two arguments:

- The first argument specifies a pointer to a string (an array of characters terminated by a null character). In this example, this argument is the controller name pointed to by the `ctlr_name` field of the pointer to the `device` structure.
- The second argument also specifies a pointer to a string. In the example, this argument is the controller name pointed to by the `ctlr_name` field of the pointer to the `controller` structure.

The code fragment sets up a condition statement that performs tasks that are based on the results of the comparison. Figure 5–1 shows how `strcmp` compares two sample character-string values in the code fragment. In item 1, `strcmp` compares the two controller names and returns the value 0 (zero) because the two strings were identical.

In item 2, `strcmp` returns an integer that is less than zero because the lexicographical comparison indicates that the characters in the first controller name, `fb`, come before the letters in the second controller name, `ipi`. In other words, the first pair of letters — in the same position in both strings — that do not match are `f` and `i`, and `f` is less than `i`.

Figure 5–1: Results of the strcmp Routine



ZK-0624U-AI

5.1.2 Comparing Two Strings by Using a Specified Number of Characters

To compare two strings by using a specified number of characters, call the `strncmp` routine. The following code fragment shows a call to `strncmp`:

```
.
.
register struct device *device;
.
.

if( (strncmp(device->dev_name, "rz", 2) == 0))[1]
.
.
```

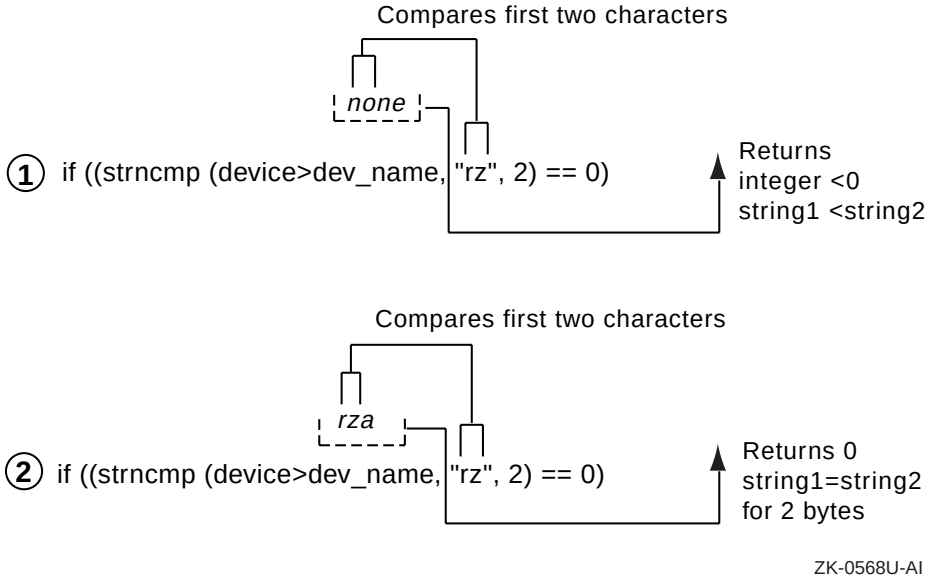
[1] Shows that the `strncmp` routine takes three arguments:

- The first argument specifies a pointer to a string. In the example, this argument is the device name pointed to by the `dev_name` field of the pointer to the `device` structure.
- The second argument also specifies a pointer to a string. In the example, this argument is the character string `rz`.
- The third argument specifies the number of bytes to be compared. In the example, the number of bytes to compare is 2.

The code fragment sets up a condition statement that performs tasks that are based on the results of the comparison. Figure 5–2 shows how `strncmp` compares two sample character-string values in the code fragment. In item 1, `strncmp` compares the first two characters of the device name `none` with the string `rz`. It then returns an integer less than the value 0 (zero), because `strncmp` makes a lexicographical comparison between the two strings and the string `no` comes before the string `rz`.

In item 2, `strncmp` compares the first two characters of the device name `rza` with the string `rz` and returns the value 0 (zero), because `strncmp` makes a lexicographical comparison between the two strings and the string `rz` is equal to the string `rz`.

Figure 5–2: Results of the strncmp Routine



5.1.3 Copying a Null-Terminated Character String

To copy a null-terminated character string, call the `strcpy` routine. The following code fragment shows a call to `strcpy`:

```
.
.
.
struct tc_slot  tc_slot[TC_IOSLOTS]; 1
char curr_module_name[TC_ROMNAMLEN + 1]; 2
.
.
.

strcpy(tc_slot[i].modulename, curr_module_name); 3
.
.
.
```

1 Declares an array of `tc_slot` structures of size `TC_IOSLOTS`.

2 Declares a variable to store the module name from the ROM of a device on the TURBOchannel bus.

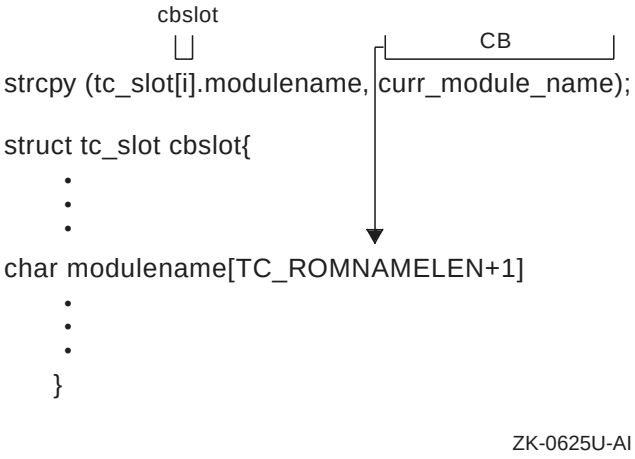
3 Shows that the `strcpy` routine takes two arguments:

- The first argument specifies a pointer to a buffer large enough to hold the string to be copied. In the example, this buffer is the `modulename` field of the `tc_slot` structure for the specified bus.
- The second argument specifies a pointer to a string. This is the string to be copied to the buffer that the first argument specifies.

In the example, this is the module name from the ROM, which is stored in the `curr_module_name` variable.

Figure 5–3 shows how `strcpy` copies a sample value in the code fragment. The routine copies the string `CB` (the value that is contained in *`curr_module_name`*) to the `modulename` field of the `tc_slot` structure for the specified bus. This field is presumed large enough to store the character string. The `strcpy` routine returns the pointer to the location following the end of the destination buffer.

Figure 5–3: Results of the strcpy Routine



ZK-0625U-AI

5.1.4 Copying a Null-Terminated Character String with a Specified Limit

To copy a null-terminated character string with a specified limit, call the `strncpy` routine. The following code fragment shows a call to `strncpy`:

```
.
.
register struct device *device;
char * buffer;
.
.

strncpy(buffer, device->dev_name, 2); 1
if (buffer == somevalue)
.
.
```

1 Shows that `strncpy` takes three arguments:

- The first argument specifies a pointer to a buffer of at least the same number of bytes as specified in the third argument. In the example, this is the pointer to the *`buffer`* variable.

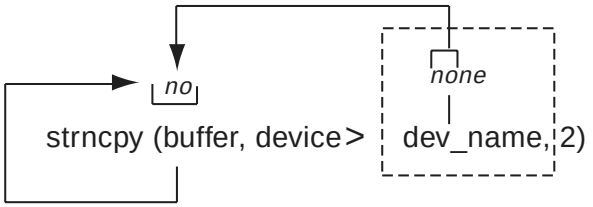
- The second argument specifies a pointer to a string. This is the character string to be copied, and in the example is the value pointed to by the `dev_name` field of the pointer to the `device` structure.
- The third argument specifies the number of characters to copy, which in the example is two characters.

The code fragment sets up a condition statement that performs some tasks that are based on the characters stored in the pointer to the *`buffer`* variable.

Figure 5–4 shows how `strncpy` copies a sample value in the code fragment. The routine copies the first two characters of the string `none` (the value pointed to by the `dev_name` field of the pointer to the `device` structure). The `strncpy` routine stops copying after it copies a null character or the number of characters that are specified in the third argument, whichever comes first.

The figure also shows that `strncpy` returns a pointer to the `/NULL` character at the end of the first string (or to the location following the last copied character if there is no NULL). The copied string will not be null terminated if its length is greater than or equal to the number of characters that are specified in the third argument.

Figure 5–4: Results of the strncpy Routine



ZK-0793U-AI

5.1.5 Returning the Number of Characters in a Null-Terminated String

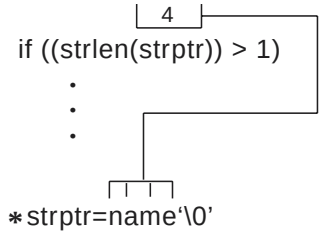
To return the number of characters in a null-terminated character string, call the `strlen` routine. The following code fragment shows a call to `strlen`:

```
.
.
.
char *strptr;
.
.
.
if ((strlen(strptr)) > 1)[1]
```

[1] Shows that the `strlen` routine takes one argument: a pointer to a string. In the example, this pointer is the variable *`strptr`*.

The code fragment sets up a condition statement that performs some tasks that are based on the length of the string. Figure 5–5 shows how `strlen` determines the number of characters in a sample string in the code fragment. As the figure shows, `strlen` returns the number of characters that the *`strptr`* variable points to, which in the code fragment is four. The `strlen` routine does not count the terminating null character.

Figure 5–5: Results of the strlen Routine



ZK-0626U-AI

5.2 Using Data Copying Routines

The data copying routines allow kernel modules to:

- Copy a series of bytes with a specified limit (Section 5.2.1)
- Zero a block of kernel memory (Section 5.2.2)
- Zero a block of memory in user space (Section 5.2.3)
- Copy data from user address space to kernel address space (Section 5.2.4)
- Copy data from kernel address space to user address space (Section 5.2.5)
- Move data between user virtual space and system virtual space (Section 5.2.6)

The following sections describe the routines that perform these tasks.

5.2.1 Copying a Series of Bytes with a Specified Limit

To copy a series of bytes with a specified limit, call the `bcopy` routine. The following code fragment shows a call to `bcopy`:

```
.
.
.
struct tc_slot  tc_slot[TC_IOSLOTS]; 1
.
.
.

char *cp; 2
.
.
.

bcopy(tc_slot[index].modulename, cp, TC_ROMNAMLEN + 1); 3
```

⋮

1 Declares an array of `tc_slot` structures of size `TC_IOSLOTS`.

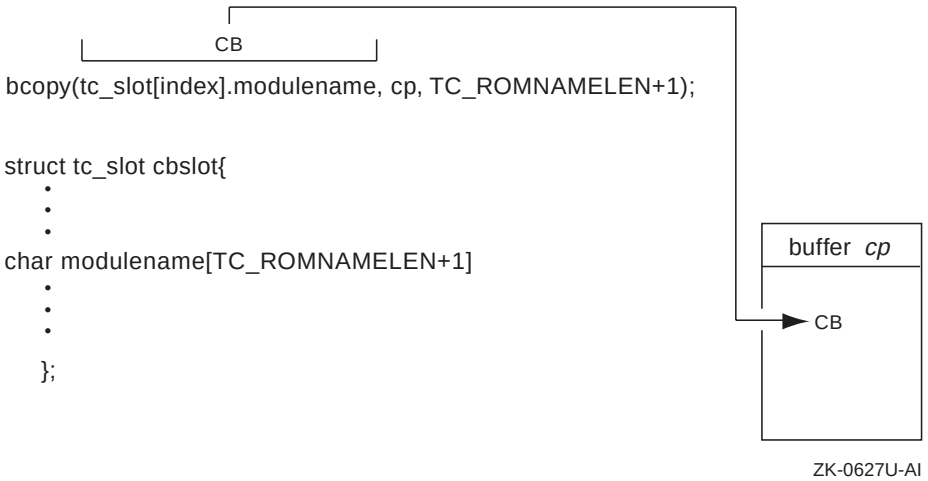
2 Declares a pointer to a buffer that stores the bytes of data that are copied from the first argument.

3 Shows that the `bcopy` routine takes three arguments:

- The first argument is a pointer to a byte string (an array of characters). In the example, this array is the `modulename` field of the `tc_slot` structure for this bus.
- The second argument is a pointer to a buffer that is at least the size that is specified in the third argument. In the example, this buffer is represented by the pointer to the *cp* variable.
- The third argument is the number of bytes to be copied. In the example, the number of bytes is the value of the constant `TC_ROMNAMLEN` plus 1.

Figure 5–6 shows how `bcopy` copies a series of bytes by using a sample value in the code fragment. As the figure shows, `bcopy` copies the characters `CB` to the buffer *cp* without searching for null bytes. The copy is nondestructive; that is, the address ranges of the first two arguments can overlap.

Figure 5–6: Results of the bcopy Routine



5.2.2 Zeroing a Block of Memory in Kernel Address Space

To zero a block of memory in kernel address space, call the `bzero` routine. The following code fragment shows a call to `bzero`:

```
.
.
.
struct bus *new_bus
.
.
.

bzero(new_bus, sizeof(struct bus)); 1
.
.
.
```

1 Shows that the `bzero` routine takes two arguments:

- The first argument is a kernel address at which the zeroing operation starts. In the example, the first argument is a pointer to a `bus` structure.
- The second argument is the number of bytes to be zeroed. In the example, this size is expressed through the use of the `sizeof` operator, which returns the size of a `bus` structure.

In the example, `bzero` is used to zero the number of bytes that are associated with the size of the `bus` structure, starting at the address specified by *new_bus*.

5.2.3 Zeroing a Block of Memory in User Address Space

To zero a block of memory in user address space, call the `uzero` routine. The following code fragment shows a call to `uzero`:

```
.
.
.
void *user_addr
size_t cnt;
.
.
.

int err;
.
.
.

if (err = uzero(user_addr, cnt)) 1
.
.
.
```

1 Shows that the `uzero` routine takes two arguments:

- The first argument is a user address at which the zeroing operation starts.
- The second argument is the number of bytes to be zeroed.

In the example, `uzero` is used to zero `cnt` bytes starting at address `user_addr`. It returns the value 0 (zero) upon successful completion. If the address in user address space cannot be accessed, `uzero` returns the error `EFAULT`.

5.2.4 Copying Data from User Address Space to Kernel Address Space

To copy data from the unprotected user address space to the protected kernel address space, call the `copyin` routine. The following code fragment shows a call to `copyin`:

```
.
.
.
struct buf *bp;
int err;
void* buff_addr;
void* kern_addr;
.
.
.
if (err = copyin(buff_addr, kern_addr, bp->b_resid)) {[1]
.
.
.
```

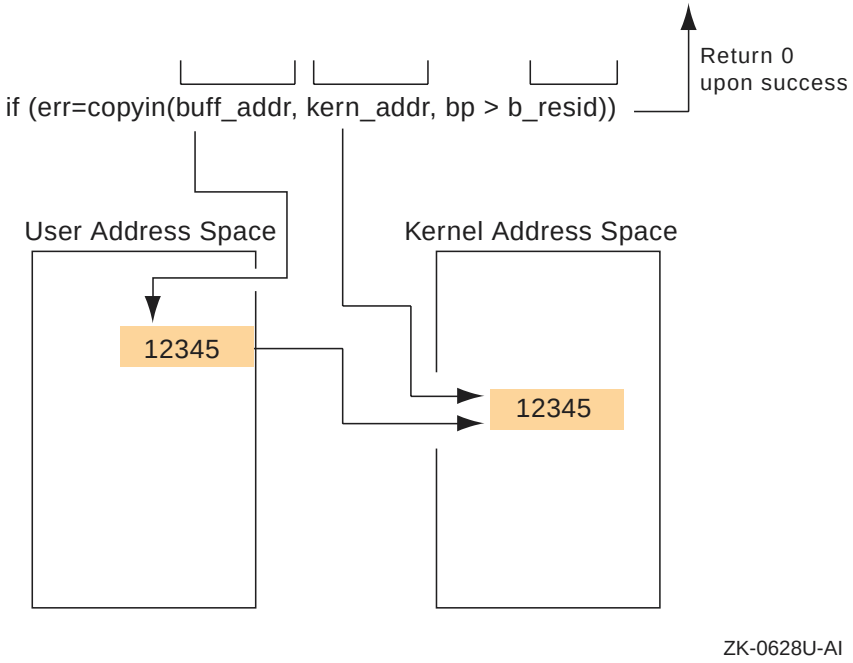
[1] Shows that the `copyin` routine takes three arguments:

- The first argument specifies the address in user space of the data to be copied. In the example, this address is the user buffer's address.
- The second argument specifies the address in kernel space to which to copy the data. In the example, this address is the address of the kernel buffer.
- The third argument specifies the number of bytes to copy. In the example, the number of bytes is contained in the `b_resid` field of the pointer to the `buf` structure.

The code fragment sets up a condition statement that performs tasks that are based on whether `copyin` executes successfully. Figure 5–7 shows how `copyin` copies data from user address space to kernel address space by using sample data.

As Figure 5–7 shows, `copyin` copies the data from the unprotected user address space (specified by *`buff_addr`*) to the protected kernel address space (specified by *`kern_addr`*). The `b_resid` field indicates the number of bytes. The figure also shows that `copyin` returns the value 0 (zero) upon successful completion. If the address in user address space cannot be accessed, `copyin` returns the error `EFAULT`.

Figure 5–7: Results of the copyin Routine



5.2.5 Copying Data from Kernel Address Space to User Address Space

To copy data from the protected kernel address space to the unprotected user address space, call the `copyout` routine. The following code fragment shows a call to `copyout`:

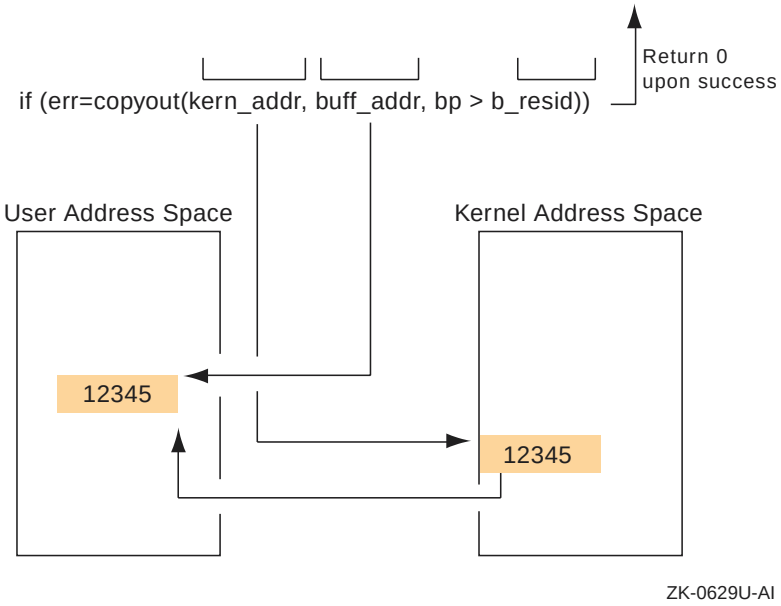
```
.
.
register struct buf *bp;
int err;
void * buff_addr;
void * kern_addr;
.
.

if (err = copyout(kern_addr,buff_addr,bp->b_resid)) {[1]
.
.
```

[1] Shows that the `copyout` routine takes three arguments:

- The first argument specifies the address in kernel space of the data to be copied. In the example, this address is the kernel buffer's address, which is stored in the *kern_addr* argument.
- The second argument specifies the address in user space to which to copy the data. In the example, this address is the user buffer's virtual address, which is stored in the *buff_addr* argument.
- The third argument specifies the number of bytes to copy. In the example, the number of bytes is contained in the `b_resid` field of the pointer to the `buf` structure.

Figure 5–8 shows the results of `copyout`, based on the code fragment. As the figure shows, `copyout` copies the data from the protected kernel address space (specified by *kern_addr*) to the unprotected user address space (specified by *buff_addr*). The number of bytes is indicated by the `b_resid` field. The figure also shows that `copyout` returns the value 0 (zero) upon successful completion. If the address in kernel address space cannot be accessed or if the number of bytes to copy is invalid, `copyout` returns the error `EFAULT`.

Figure 5–8: Results of the copyout Routine



5.2.6 Moving Data Between User Virtual Space and System Virtual Space

To move data between user virtual space and system virtual space, call the `uiomove` routine. The following code fragment shows a call to `uiomove`:

```
.
.
.
struct uio *uio;
void * kern_addr;
int err;
long cnt;
.
.
.

err = uiomove(kern_addr,cnt,uio); 1
.
.
.
```

1 Shows that the `uiomove` routine takes three arguments:

- The first argument specifies a pointer to the kernel buffer in system virtual space.
- The second argument specifies the number of bytes of data to be moved. In this example, the number of bytes to be moved is stored in the *cnt* variable.

- The third argument specifies a pointer to a `uio` structure. This structure describes the current position within a logical user buffer in user virtual space.

5.3 Using Kernel-Related Routines

The kernel-related routines allow kernel modules to:

- Print text to the console and error logger (Section 5.3.1)
- Put a calling process to sleep (Section 5.3.2)
- Wake up a sleeping process (Section 5.3.3)
- Initialize a timer (callout) queue element (Section 5.3.4)
- Remove the scheduled routine from the timer queues (Section 5.3.5)
- Set the interrupt priority mask (Section 5.3.6)
- Allocate memory (Section 5.3.7)

The following sections describe the routines that perform these tasks.

5.3.1 Printing Text to the Console and Error Logger

To print text to the console terminal and the error logger, call the `printf` routine. The kernel `printf` routine is a scaled-down version of the C library `printf` routine. The `printf` routine prints diagnostic information directly on the console terminal and writes ASCII text to the error logger. Because `printf` is not interrupt driven, all system activities are suspended when you call it. Only a limited number of characters (currently 128) can be sent to the console display during each call to `printf` because the characters are formatted into a fixed-size buffer whose address may be handed off to the primary CPU for console output. If more than 128 characters are generated in a single call to `printf`, all characters following the first 128 will be discarded.

If you need to see the results on the console terminal, limit the message size to the maximum of 128 whenever you send a message from within the module. However, `printf` also stores the messages in an error log file. You can use the `uerf` command to view the text of this error log file. See `printf`(9) for more information. The messages are easier to read if you use `uerf` with the `-o terse` option.

The following code fragment shows a call to this routine:

```
.
.
printf("CBprobe ctlr = %8x\n",ctlr);
.
.
```

The code example shows a typical use for the `printf` routine in the debugging of kernel modules. In the example, `printf` takes two arguments:

- The first argument specifies a pointer to a string that contains two types of objects. One object is ordinary characters such as, "hello, world," which are copied to the output stream. The other object is a conversion specification, such as %d. (Supported conversion specifications include %c, %d, %ld, %lx, %o, %s, and %x. See `printf`(9) for explanations of these specifications.)
- The second argument specifies the value to be formatted in place of the `%8x` specifier in the format string. In this example, the argument is *`ctlr`*.

The operating system also supports the `uprintf` routine. The `uprintf` routine prints to the current user's terminal. Never have interrupt service routines call `uprintf`. Do not use this routine to print verbose messages. The `uprintf` routine does not log messages to the error logger.

5.3.2 Putting a Calling Process to Sleep

To put a calling process to sleep in a symmetric multiprocessing (SMP) environment, call the `mpsleep` routine. The `mpsleep` routine blocks the current kernel thread until a wakeup is issued (see Section 5.3.3).

Generally, kernel modules call this routine to wait for the transfer to complete an interrupt from the device. That is, the `write` routine of the kernel module sleeps on the address of a known location, and the device's interrupt service routine wakes the process when the device interrupts. The wakened process determines whether the condition for which it was sleeping has been removed. The following code fragment shows a call to this routine:

```
.
.
mpsleep((vm_offset_t)&sc->error_recovery_flag, PCATCH,
         "ftaerr", 0, &sc->lk_fta_kern_str,
          MS_LOCK_SIMPLE | MS_LOCK_ON_ERROR)) [1]
.
.
```

[1] Calls the `mpsleep` routine to block the current kernel thread. The `mpsleep` routine takes several arguments:

- The `channel` argument specifies an address to associate with the calling kernel thread to be put to sleep. In this example, the address

(or event) associated with the current kernel thread is stored in the `error_recovery_flag` field.

- The `pri` argument specifies whether the sleep request is interruptible. Setting this argument to the `PCATCH` option causes the process to sleep in an interruptible state (that is, the kernel thread can take asynchronous signals). Not setting the `PCATCH` option causes the process to sleep in an uninterruptible state (that is, the kernel thread cannot take asynchronous signals).
- The `wmesg` argument specifies the wait message. In this call, `fta_error_recovery` passes the string `ftaerr`.
- The `timo` argument specifies the maximum amount of time that the kernel thread should block. If you pass the value 0 (zero), `mpsleep` assumes there is no timeout.
- The `lockp` argument specifies a pointer to a simple or complex lock. You pass a simple or complex lock structure pointer if you want to release the lock. Pass the value 0 (zero) if you do not want to release the lock.
- The `flags` argument specifies the lock type. You can pass the bitwise inclusive OR of the valid lock bits defined in `/usr/sys/include/sys/param.h`.

.

5.3.3 Waking Up a Sleeping Process

To wake up all processes that are sleeping on a specified address, call the `wakeup` routine. The following code fragment shows a call to this routine:

```
.
.
wakeup(&ctlr->bus_name); 1
.
.
```

1 Shows that the `wakeup` routine takes one argument: the address on which the wakeup is to be issued. In the example, this address is the bus name for the bus to which this controller is connected. This address was specified in a previous call to the `mpsleep` routine. All processes that are sleeping on this address are awakened.

5.3.4 Initializing a Timer (Callout) Queue Element

To initialize a timer queue element, call the `timeout` routine. The following code fragment shows a call to this routine:

```
.
.
.
#define NONEIncSec  1
.
.
.
cb = &none_unit[unit];
.
.
.
timeout(noneincled, (caddr_t)none, NONEIncSec*hz); 1
.
.
.
```

1 Shows that the `timeout` routine takes three arguments:

- The first argument specifies a pointer to the routine to be called. In the example, `timeout` will call the `noneincled` routine on the interrupt stack (not in processor context) as dispatched from the `softclock` routine.
- The second argument specifies a single argument to be passed to the called routine. In the example, this argument is the pointer to the `NONE` device's `none_unit` data structure. This argument is passed to the `noneincled` routine. Because the data types of the arguments are different, the code fragment performs a type-casting operation that converts the argument type to be of type `caddr_t`.
- The third argument specifies the amount of time to delay before calling the specified routine. You express time as **ticks** of the system clock. To obtain a particular time in seconds, you multiply the number of ticks times `hz` (`hz` contains the number of ticks per second).

  In the example, the constant `NONEIncSec` is used with the *hz* global variable to determine the amount of time before `timeout` calls `noneincled`. The global variable *hz* contains the number of clock ticks per second. This variable is a second's worth of clock ticks. The example shows a 1-second delay.

5.3.5 Removing Scheduled Routines from the Timer (Callout) Queue

To remove the scheduled routines from the timer queue, call the `untimeout` routine. The following code fragment shows a call to this routine:

```
.
.
.
untimeout(noneincled, (caddr_t)none); 1
.
.
.
```

1 Shows that the `untimeout` routine takes two arguments:

- The first argument specifies a pointer to the routine to be removed from the timer queue. In the example, `untimeout` removes the `noneincled` routine from the timer queue. This routine was placed on the timer queue in a previous call to the `timeout` routine.
- The second argument specifies a single argument to be passed to the called routine. In the example, this argument is the pointer to the `NONE` device's `none_unit` data structure. It matches the parameter that was passed in a previous call to `timeout`. Because the data types of the arguments are different, the code fragment performs a type-casting operation that converts the argument type to be of type `caddr_t`.

The two arguments uniquely identify which timeout entry to remove. This is useful if more than one thread has called `timeout` with the same routine argument.

5.3.6 Setting the Interrupt Priority Mask

To set the interrupt priority level (IPL) mask to a specified level, call one of the `spl` routines. Table 5–1 summarizes the uses for the different `spl` routines.

Table 5–1: Uses for spl Routines

spl Routine	Meaning
`splextreme`	Highest priority; blocks everything except halt interrupts (for example, real-time devices, machine checks, and so forth).
`splrt`	Blocks real-time devices but allows machine checks and halt interrupts.
`splclock`	Masks all hardware clock and lower-level interrupts.
`splhigh`	Masks all interrupts except clock interrupts, real-time devices, machine checks, and halt interrupts.
`spldevhigh`	Masks all device and software interrupts.
`splbio`	Masks all disk and tape controller interrupts.
`splimp`	Masks all LAN hardware interrupts.
`splvm`	Masks all interrupts that affect virtual memory operations.
`splnet`	Masks all network software interrupts.
`splsoftclock`	Masks all software clock interrupts.

Table 5–1: Uses for spl Routines (cont.)

spl Routine	Meaning
splx	Resets the CPU priority to the level specified by the argument.
splnone	Unmasks (enables) all interrupts.

The spl routines set the CPU priority to various interrupt levels. The current CPU priority level determines which types of interrupts are masked (disabled) and which are unmasked (enabled). Historically, seven levels of interrupts were supported, with eight different spl routines to handle the possible cases. For example, calling spl0 unmasked all interrupts and calling spl7 masked all interrupts. Calling an spl routine between 0 and 7 masked all interrupts at that level and at all lower levels.

Specific interrupt levels were assigned for different device types. For example, before it handled a given interrupt, a kernel module set the CPU priority level to mask all other interrupts of the same level or lower. This setting meant that the kernel module could be interrupted only by interrupt requests from devices of a higher priority.

The operating system currently supports the naming of spl routines to indicate the associated device types. Named spl routines make it easier to determine which routine to use to set the priority level for a given device type.

The following code fragment shows the use of spl routines as part of a disk strategy routine:

```
.
.
int s;
.
.

s = splbio(); 1
.
.

[Code to deal with data that can be modified by the disk interrupt
code]
splx(s); 2
.
.
```

1 Calls the splbio routine to mask (disable) all disk interrupts. This routine does not take an argument.

2 Calls the splx routine to reset the CPU priority to the level that the *s* argument specifies. The argument associated with splx is a CPU priority level, which in the example is the value that splbio returns. (The splx routine is the only one of the spl routines that takes an

argument.) Upon successful completion, each spl routine returns an integer value that represents the CPU priority level that existed before it was changed by a call to the specified spl routine.

5.3.7 Allocating Memory

A kernel module may need to declare a significant number of data structures to contain a large amount of data. For example, a kernel module that is a device driver may need to support a large number of disks and controllers. Statically allocating the maximum number of data structures wastes space. Dynamically allocating memory for the required data structures is a better use of system resources, especially when working with temporary or transient data.

To dynamically allocate memory, you need to:

- Use the MALLOC macro to allocate the data structures
- Use the FREE macro to free up the dynamically allocated data structures

The following sections describe these steps.

5.3.7.1 Allocating Data Structures with MALLOC

Use the MALLOC macro to dynamically allocate a variable-size section of kernel virtual memory. The MALLOC macro maintains a pool of preallocated memory for quick allocation and returns the address of the allocated memory. The MALLOC macro is actually a wrapper that calls malloc. Do not allow a kernel module to directly call the malloc routine.

The syntax for the MALLOC macro is as follows:

```
MALLOC(
       addr,
       cast,
       u_long size,
       int type,
       int flags );
```

Call the MALLOC macro with the following parameters:

addr

Specifies the memory location that points to the allocated memory. You specify the addr argument's data type in the cast argument.

cast

Specifies the data type of the addr argument and the type of the memory pointer that MALLOC returns.

size

Specifies the size, in byte,s of the memory to allocate. Typically, you pass the size as a constant to speed up the memory allocation.

type

Specifies the purpose for which the memory is being allocated. The memory types are defined in the `sys/malloc.h` file. Typically, kernel modules use the constant `M_DEVBUF` to indicate that kernel module memory is being allocated (or freed).

flags

Specifies one of the following constants that are defined in `/usr/sys/include/sys/malloc.h`:

`M_WAITOK`	Allocates memory from the virtual memory subsystem if there is not enough memory in the preallocated pool. This constant signifies that `MALLOC` can block.
`M_NOWAIT`	Does not allocate memory from the virtual memory subsystem if there is not enough memory in the preallocated pool. This constant signifies that `MALLOC` cannot block. `M_NOWAIT` must be used when calling `MALLOC` from an interrupt context or if the caller is holding a simple lock. Otherwise, a system panic will occur.
`M_ZERO`	Allocates zero-filled memory. You pass this bit value using the `OR` operator with `M_WAITOK` or `M_NOWAIT`.

The following example shows how to allocate memory using the `MALLOC` macro:

```
struct foo *foo1;
struct foo *foo2;
struct bar *bar[];
.
.
.
MALLOC(foo1, struct foo *, sizeof(struct foo),
 M_DEVBUF, M_NOWAIT|M_ZERO);1

if (!foo1) {
.
.
.
```

```
 return; 2
}
.
.
.
 MALLOC(foo2, struct foo *,
    nfoo * sizeof(struct foo), M_DEVBUF,
    M_WAITOK|M_ZERO); 3
.
.
.
MALLOC(bar, struct bar **,
    nbar * sizeof(struct bar *), M_DEVBUF,
    M_WAITOK|M_ZERO); 4
.
.
.
MALLOC(bar[1], struct bar *, sizeof(struct bar),
   M_DEVBUF, M_WAITOK|M_ZERO); 5
```

1 Allocates a single data structure.

2 Because M_NOWAIT is specified, examines the return value to determine whether the allocation failed.

3 Allocates an array of structures with nfoo elements.

4 Allocates an array of pointers to structures.

5 Allocates a structure to the second element of bar.

5.3.7.2 Freeing Up Dynamically Allocated Memory

When a block of memory that is allocated through MALLOC is no longer needed it, free it back to the system using the FREE macro. The FREE macro takes two arguments:

- The first argument specifies the memory pointer that points to the allocated memory to be freed. You must have previously set this argument in the call to MALLOC.
- The second argument specifies the purpose for which the memory is being allocated. The memory types are defined in the file /usr/sys/include/sys/malloc.h. Typically, kernel modules that are device drivers use the constant M_DEVBUF to indicate that memory is being allocated (or freed).

The following example shows how to use the FREE macro:

```
FREE(foo1, M_DEVBUF);

/*
 * Free the second element from the array of pointers
 */
FREE(bar[1], M_DEVBUF);
bar[1] = NULL;
```

5.4 Working with System Time

This section describes considerations for working with system time. Information in this section explains the following concepts:

- Understanding system time concepts (Section 5.4.1)
- Fetching time (Section 5.4.2)
- Modifying a timestamp (Section 5.4.3)
- Enabling an application to convert time to a string (Section 5.4.4)
- Delaying a routine a specified number of microseconds (Section 5.4.5)

5.4.1 Understanding System Time Concepts

This section discusses concepts for working with system time:

- How a kernel module fetches or modifies time
- How time is created

5.4.1.1 How a Kernel Module Uses Time

Kernel modules can save timestamps that can be passed to applications on request for many purposes. For example:

- When a bus was last scanned
- When the last error on a disk occurred
- When the last interrupt for the some device (for example, a line printer) occurred
- When the system booted
- When the file system was mounted on a particular disk

The application then needs to print the date and time. Your kernel module code must determine several things for each timestamp that it wants to preserve:

- When it needs to fetch time
- Whether or not the time value that was fetched needs modification to reflect accurate time
- How to pass the time value to the application

5.4.1.2 How System Time is Created

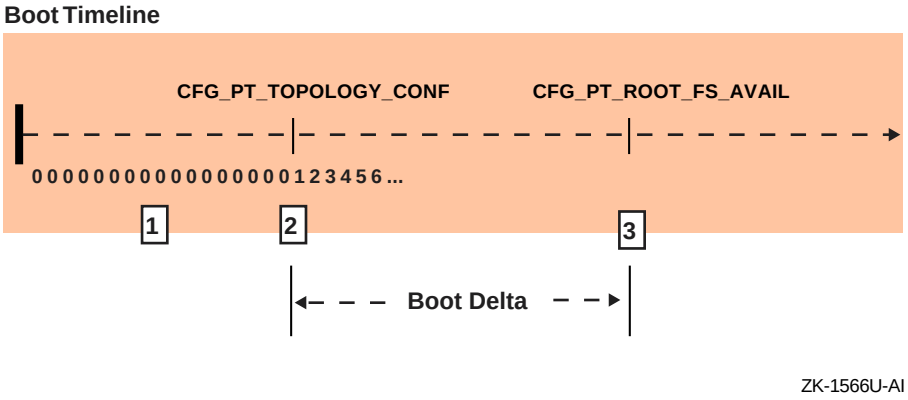
System time, which is platform-dependent, is defined as ticks of the system clock, measured as units of hertz (hz). The operating system makes system time available to kernel modules. The representation of system time is not

based on the current calendar time of day because the actual time value does not become available to the operating system until you are partially through the boot sequence.

From the beginning of a boot sequence to dispatch point CFG_PT_TOPOLOGY_CONF, the operating system time value is 0 (zero). In Tru64 UNIX, zero is equivalent to January 1, 1970, 00:00:00, UTC. At dispatch point CFG_PT_TOPOLOGY_CONF, the operating system begins incrementing system time from zero. Later, at the dispatch point CFG_PT_ROOT_FS_AVAIL, system time is set to the actual time of day.

The time between CFG_PT_TOPOLOGY_CONF and CFG_PT_ROOT_FS_AVAIL is called the **boot delta**. Figure 5–9 shows these concepts.

Figure 5–9: When Time Becomes Available During a System Boot



1. At the start of a boot sequence, the value is 0 (zero).
2. At CFG_PT_TOPOLOGY_CONF, the kernel starts incrementing time. The initial date and time is recorded as 00:00:00 UTC 1 Jan 1970 (the Epoch).
3. At CFG_PT_ROOT_FS_AVAIL, the kernel sets the time to the correct calendar date and time.

If your kernel module fetches time before CFG_PT_ROOT_FS_AVAIL is reached, the time value it fetches is incorrect and you will need to modify that timestamp later (see Section 5.4.3).

5.4.2 Fetching System Time

A kernel module decides when to fetch system time. When it performs a fetch operation, it also needs a way to fetch system time. The TIME_READ macro provides a way for your kernel module to fetch the current time. The following code fragment shows how to use this macro in your kernel module:

```
#include <sys/time.h> 1
.
.
.
extern struct timeval time; 2
.
.
.
{   struct timeval my_time; 3
    .
    .
    .
    TIME_READ(my_time); 4
```

1 Includes the `time.h` header file.

2 Declares the global time variable as external.

3 Declares your own storage for your timestamp.

4 Fetches the current time and stores it in your own time variable using the `TIME_READ` macro. `TIME_READ` takes one parameter, which specifies the memory location to store the current time. Its type is `struct timeval`.

5.4.3 Modifying a Timestamp

If your kernel module fetches time before the operating system sets the current time at `CFG_PT_ROOT_FS_AVAIL`, you must modify the timestamp you fetched and stored. For example, assume your kernel module keeps track of when it last scanned the bus. Because scanning the bus takes place prior to `CFG_PT_ROOT_FS_AVAIL`, the fetched time is interpreted as approximately Jan. 1, 1970, 00:00:00. (This is because time was not set to the proper value when you fetched it.) The global variable `bootdelta` keeps track of how many seconds and microseconds have been counted between the two configuration points.

To modify a timestamp, follow these steps:

1. Register a callback for `CFG_PT_ROOT_FS_AVAIL` in your kernel module.
2. Use the following algorithm to modify the timestamp:
   a. Subtract the number of seconds (`tv_sec`) and microseconds (`tv_usec`) that were counted before time was set to the actual time.
   b. Add the number of seconds and microseconds that were counted to the point where the kernel module fetched time.

The following code example subtracts `bootdelta` seconds and adds `my_time` seconds:

```
#include <sys/time.h>
.
.
.
```

```
extern struct timeval bootdelta;
.
.

struct timeval temp_time;
  TIME_READ(temp_time); [1]
.
.

   temp_time.tv_sec -= (bootdelta.tv_sec - my_time.tv_sec); [2]

   if (bootdelta.tv_usec > temp_time.tv_usec) {
    temp_time.tv_usec = 1000000 -
      (bootdelta.tv_usec - temp_time.tv_usec);
    temp_time.tv_sec--;
  } else {
    temp_time.tv_usec -= bootdelta.tv_usec; [3]
  }
.
.

    temp_time.tv_usec += my_time.tv_usec; [4]

    if (temp_time.tv_usec >= 1000000) {
      temp_time.tv_usec -= 1000000;
      temp_time.tv_sec++; [5]
  }
.
.

  my_time = temp_time; [6]
```

[1] Obtains the current time, which is set to the actual time of day.

[2] Subtracts `bootdelta` seconds from the current time and adds the number of seconds in the timestamp.

[3] Subtracts `bootdelta` microseconds; make sure its value is not negative.

[4] Adds `my_time` microseconds.

[5] Fixes any microseconds that may have wrapped.

[6] Stores the results into the time variable.

5.4.4 Enabling Applications to Convert a Kernel Timestamp to a String

A user application can receive a timestamp from a kernel module in a variety of ways. The standard way is for a kernel module to pass a timestamp to the application as a `struct timeval`.

For an application to convert the timestamp it received from the kernel module, it uses the `ctime` function that is defined in `/usr/include/sys/time.h`. This function converts time values between `tm` structures, `time_t` type variables, and strings.

The ctime function expresses time in units by converting the time_t variable, to which thetimer parameter points, into a string with the five-field format. The time_t variable, which is also defined in /usr/include/sys/time.h, contains the number of seconds since the Epoch, 00:00:00 UTC 1 Jan 1970. For example:

```
Tue Jul 11 15:37:29 2000
```

For more information on converting timestamps to strings, see ctime(3).

5.4.5 Delaying the Calling Routine a Specified Number of Microseconds

To delay the calling routine a specified number of microseconds, use the DELAY macro. The following code fragment shows how to use this macro:

```
.
.
.
DELAY(10000) 1
.
.
.
```

1 Shows that the DELAY macro takes one argument: the number of microseconds for the calling thread to spin.

The DELAY macro delays the routine by a specified number of microseconds. DELAY spins while it waits for the specified number of microseconds to pass before continuing execution. The example shows a 10000-microsecond (10-millisecond) delay. The range of delays is system dependent, due to its relation to the granularity of the system clock. The system defines the number of clock ticks per second in the *hz* variable. Specifying any value smaller than 1/hz to the DELAY macro results in an unpredictable delay. For any delay value, the actual delay may vary by plus or minus one clock tick.

We do not recommend using the DELAY macro because the processor will be consumed for the specified time interval; therefore it will be unavailable to service other threads. In cases where kernel modules need timing mechanisms, use the sleep and timeout routines instead of the DELAY macro. The most common usage of the DELAY macro is in the system boot path. Using DELAY in the boot timeline is often acceptable because there are no other threads in contention for the processor.

5.5 Using Kernel Threads

A kernel thread is a single sequential flow of control within a kernel module or other systems-based program. The kernel module or other systems-based program makes use of the routines (instead of a threads library package

such as POSIX Threads Library) to start, terminate, and delete threads, and to perform other kernel thread operations.

Kernel threads execute within (and share) a single address space. Therefore, kernel threads read and write to the same memory locations.

You use kernel threads to improve the performance of a kernel module. Multiple kernel threads are useful in a multiprocessor environment, where kernel threads run concurrently on separate CPUs. However, multiple kernel threads also improve kernel module performance on single-processor systems by permitting the overlap of input, output, or other slow operations with computational operations.

Kernel threads allow kernel modules to perform other useful work while waiting for a device to produce its next event, such as the completion of a disk transfer or the receipt of a packet from the network. For more information on using kernel threads, see Chapter 9.

5.6 Using Locks

In a single-processor environment, kernel modules need not protect the integrity of a resource from activities that result from the actions of another CPU. However, in a symmetric multiprocessing (SMP) environment, the kernel module must protect (lock) the resource from multiple CPU access to prevent corruption. A resource, from the kernel module's standpoint, is data that more than one kernel thread can manipulate. Locks are the mechanism for sharing resources in an SMP enviroment.

See Chapter 6 for an overview of symmetric multiprocessing and the two locking methods that you can use when your kernel modules execute in an SMP environment. Chapter 7 provides information for using simple locks in your kernel module. Chapter 8 provides information for using complex locks.

6

Symmetric Multiprocessing and Locking Methods

Symmetric multiprocessing (SMP) describes a computer environment that uses two or more central processing units (CPUs). In an SMP environment, software applications and the associated kernel modules can operate on two or more of these CPUs simultaneously. To ensure the integrity of the data manipulated by kernel modules in this multiprocessor environment, you must perform additional design and implementation tasks beyond those discussed in *Writing Device Drivers*. One of these tasks involves choosing a locking method. Tru64 UNIX provides you with two methods to write SMP-safe kernel modules: **simple locks** and **complex locks**.

This chapter presents information that will help you decide which items (variables, data structures, and code blocks) must be locked in the kernel module and then choose the appropriate locking method (simple or complex). Specifically, the chapter describes the following topics associated with designing and developing a kernel module that can operate safely in an SMP environment:

- Understanding hardware issues related to synchronization (Section 6.1)
- Understanding the need for locking in an SMP environment (Section 6.2)
- Comparing simple locks and complex locks (Section 6.3)
- Choosing a locking method (Section 6.4)
- Choosing the resources to lock in a kernel module (Section 6.5)

The following sections discuss each of these topics. You do not need an intimate understanding of kernel threads to learn about writing kernel modules in an SMP environment. Chapter 9 discusses kernel threads and the associated routines that kernel modules use to create and manipulate them.

6.1 Understanding Hardware Issues Related to Synchronization

Alpha CPUs provide several features to assist with hardware-level synchronization. Even though all instructions that access memory are noninterruptible, no single one performs an atomic read-modify-write

operation. A kernel-mode thread of execution can raise the interrupt priority level (IPL) to block other kernel threads on that CPU while it performs a read-modify-write sequence or while it executes any other group of instructions. Code that runs in any access mode can execute a sequence of instructions that contains load-locked (LDx_L) and store-conditional (STx_C) instructions to perform a read-modify-write sequence that appears atomic to other kernel threads of execution.

Memory barrier instructions order a CPU's memory reads and writes from the viewpoint of other CPUs and I/O processors. The locking mechanisms (simple and complex locks) that are provided in the operating system take care of the idiosyncracies that are related to read-modify-write sequences and memory barriers on Alpha CPUs. Therefore, you need not be concerned about these hardware issues when you implement SMP-safe kernel modules that use simple and complex locks.

The rest of this section describes the following hardware-related issues:

- Atomicity (Section 6.1.1)
- Alignment (Section 6.1.2)
- Granularity (Section 6.1.3)

6.1.1 Atomicity

Software synchronization refers to the coordination of events so that only one event happens at a time. This kind of synchronization is a serialization or sequencing of events. Serialized events are assigned an order and processed one at a time in that order. While a serialized event is being processed, no other event in the series is allowed to interrupt it.

By imposing order on events, software synchronization allows reading and writing of several data items indivisibly, or atomically, to obtain a consistent set of data. For example, all of process A's writes to shared data must happen before or after process B's writes or reads, but not during process B's writes or reads. In this case, all of process A's writes must happen indivisibly for the operation to be correct. This includes process A's updates — reading of a data item, modifying it, and writing it back (read-modify-write sequence). Other synchronization techniques ensure the completion of an asynchronous system service before the caller tries to use the results of the service.

Atomicity is a type of serialization that refers to the indivisibility of a small number of actions, such as those that occur during the execution of a single instruction or a small number of instructions. With more than one action, no single action can occur by itself. If one action occurs, then all the actions occur. Atomicity must be qualified by the viewpoint from which the actions appear indivisible: an operation that is atomic for kernel threads that run on

the same CPU can appear as multiple actions to a kernel thread of execution that runs on a different CPU.

An atomic memory reference results in one indivisible read or write of a data item in memory. No other access to any part of that data can occur during the course of the atomic reference. Atomic memory references are important for synchronizing access to a data item that is shared by multiple writers or by one writer and multiple readers. References need not be atomic to a data item that is not shared or to one that is shared but is only read.

6.1.2 Alignment

Alignment refers to the placement of a data item in memory. For a data item to be naturally aligned, its lowest-addressed byte must reside at an address that is a multiple of the size of the data item (in bytes). For example, a naturally aligned longword has an address that is a multiple of 4. The term *naturally aligned* is usually shortened to "aligned."

An Alpha CPU allows atomic access only to an aligned longword or an aligned quadword. Reading or writing an aligned longword or quadword of memory is atomic with respect to any other kernel thread of execution on the same CPU or on other CPUs.

6.1.3 Granularity

Granularity of data access refers to the size of neighboring units of memory that can be written independently and atomically by multiple CPUs. Regardless of the order in which the two units are written, the results must be identical.

Alpha systems have longword and quadword granularity. That is, only adjacent aligned longwords or quadwords can be written independently. Because Alpha systems support only instructions that load or store longword-sized and quadword-sized memory data, the manipulation of byte-sized and word-sized data on Alpha systems requires that the entire longword or quadword that contains the byte-sized or word-sized item be manipulated. Therefore, simply because of its proximity to an explicitly shared data item, neighboring data might become shared unintentionally. Manipulation of byte-sized and word-sized data on Alpha systems requires multiple instructions that:

1. Fetch the longword or quadword that contains the byte or word
2. Mask the nontargeted bytes
3. Manipulate the target byte or word
4. Store the entire longword or quadword

Because this sequence is interruptible, operations on byte and word data are not atomic on Alpha systems. Also, this change in the granularity of memory access can affect the determination of which data is actually shared when a byte or word is accessed.

The absence of byte and word granularity on Alpha systems has important implications for access to shared data. In effect, any memory write of a data item other than an aligned longword or quadword must be done as a multiple-instruction read-modify-write sequence. Also, because the amount of data read and written is an entire longword or quadword, you must ensure that all accesses to fields within the longword or quadword are synchronized with each other.

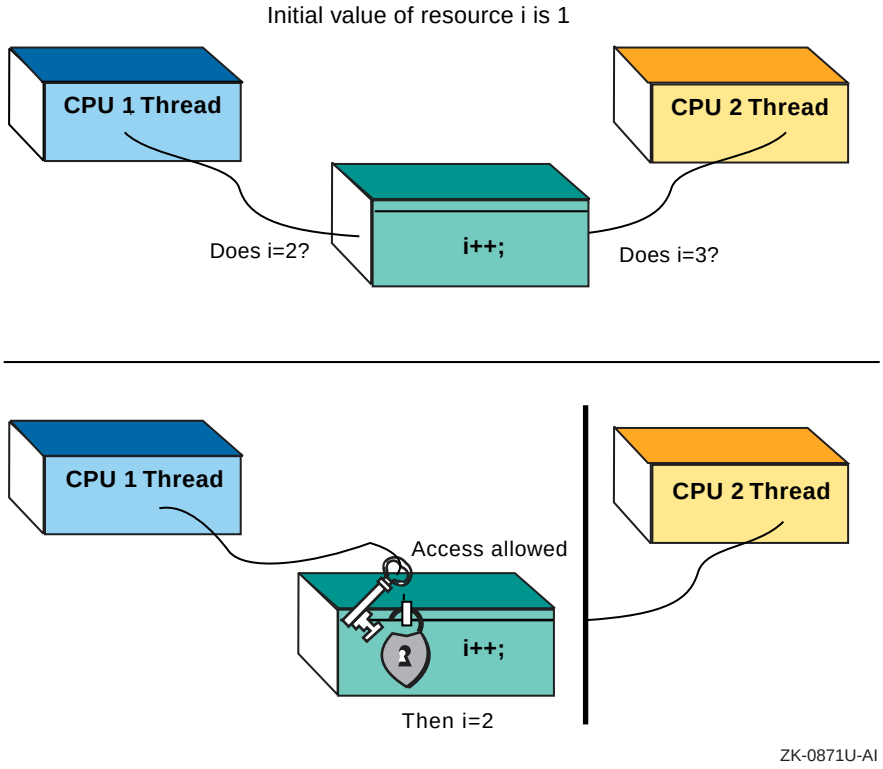
6.2 Locking in a Symmetric Multiprocessing Environment

In a single-processor environment, kernel modules do not need to protect the integrity of a resource from activities that result from the actions of another CPU. However, in an SMP environment, the kernel module must protect the resource from multiple CPU access to prevent corruption. A resource, from the kernel module's standpoint, is data that more than one kernel thread can manipulate. You can store the resource in variables (global) and in data structure fields. The top half of Figure 6–1 shows a typical problem that can occur in an SMP environment. The figure shows that the resource called `i` is a global variable whose initial value is 1.

Furthermore, the figure shows that the kernel threads from CPU1 and CPU2 increment resource `i`. A kernel thread is a single sequential flow of control within a kernel module or other systems-based program. The kernel module or other systems-based program makes use of the routines (instead of a threads library package such as the POSIX Threads Library) to start, terminate, delete, and perform other kernel threads–related operations. These kernel threads cannot increment this resource simultaneously. By locking the global variable when one kernel thread is incrementing it, you ensure that the integrity of the data that is stored in this resource is not compromised in the SMP environment.

To protect the integrity of the data, you must enforce order on the accesses of the data by multiple CPUs. One way to establish the order of CPU access to the resource is to establish a lock. As the bottom half of the figure shows, the kernel thread from CPU1 locks access to resource `i`, which prevents access by kernel threads from CPU2. This guarantees the integrity of the value stored in this resource.

Figure 6–1: Why Locking Is Needed in an SMP Environment



The vertical line in the bottom half of the figure represents a barrier that prevents the kernel thread from CPU2 from accessing resource `i` until the kernel thread from CPU1 unlocks it. For simple locks, this barrier indicates that the lock is exclusive. That is, no other kernel thread can gain access to the lock until the kernel thread currently controlling it has released (unlocked) it.

For complex write locks, this barrier represents a wait hash queue that collects all of the kernel threads that are waiting to gain write access to a resource. With complex read locks, all kernel threads have read-only access to the same resource at the same time.

6.3 Comparing Simple Locks and Complex Locks

The operating system provides two ways to lock specific resources (global variables and data structures) that are referenced in code blocks in the kernel module: simple locks and complex locks. Simple and complex locks allow kernel modules to:

- Synchronize access to a resource or resources. Kernel threads from multiple CPUs can safely update the count of global variables, add elements to or delete elements from linked lists, and update or read time elements.
- Ensure a consistent view of state transitions (run to block and block to run) across multiple CPUs.
- Make the operating system behave as though it were running on a single CPU.

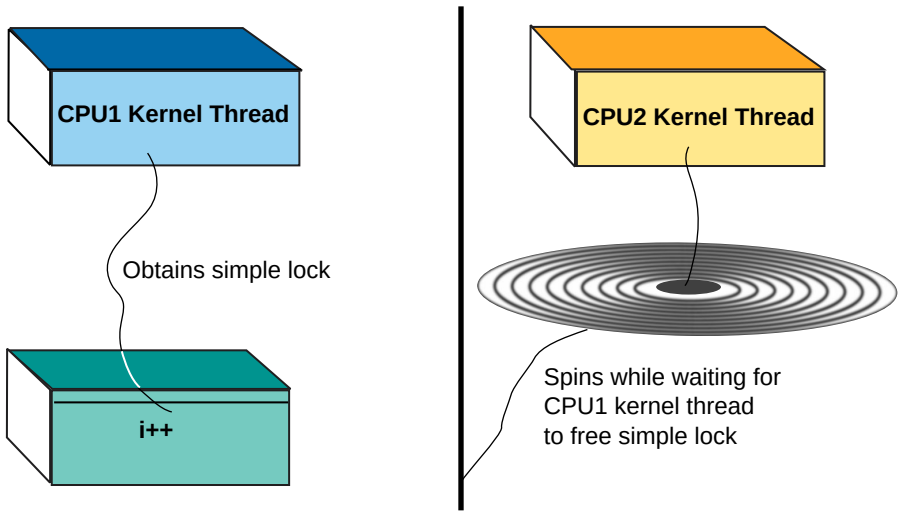
The following sections briefly describe simple locks and complex locks.

6.3.1 Simple Locks

A simple lock is a general-purpose mechanism for protecting resources in an SMP environment. Figure 6–2 shows that simple locks are spin locks. That is, the routines that implement the simple lock do not return until the lock has been obtained.

As the figure shows, the CPU1 kernel thread obtains a simple lock on resource `i`. After the CPU1 kernel thread obtains the simple lock, it has exclusive access over the resource to perform read and write operations on the resource. The figure also shows that the CPU2 kernel thread spins while waiting for the CPU1 kernel thread to unlock (free) the simple lock.

Figure 6–2: Simple Locks Are Spin Locks



You need to understand the tradeoffs in performance and real-time preemption latency that are associated with simple locks before you use them. However, kernel modules must often use simple locks. For example, kernel modules must use simple locks and `spl` routines to synchronize with interrupt service routines. Section 6.4 provides guidelines to help you choose between simple locks and complex locks.

Table 6–1 lists the data structure and routines for simple locks. Chapter 7 discusses how to use the data structure and routines to implement simple locks in a kernel module.

Table 6–1: Data Structure and Routines Associated with Simple Locks

Structure/Routines	Description
`slock`	Contains simple lock–specific information.
`decl_simple_lock_data`	Declares a simple lock structure.
`simple_lock`	Asserts a simple lock.
`simple_lock_init`	Initializes a simple lock structure.
`simple_lock_terminate`	Terminates using a simple lock.
`simple_lock_try`	Tries to assert a simple lock.
`simple_unlock`	Releases a simple lock.

6.3.2 Complex Locks

A complex lock is a mechanism for protecting resources in an SMP environment. A complex lock achieves the same results as a simple lock. However, use complex locks (not simple locks) for kernel modules if there are blocking conditions.

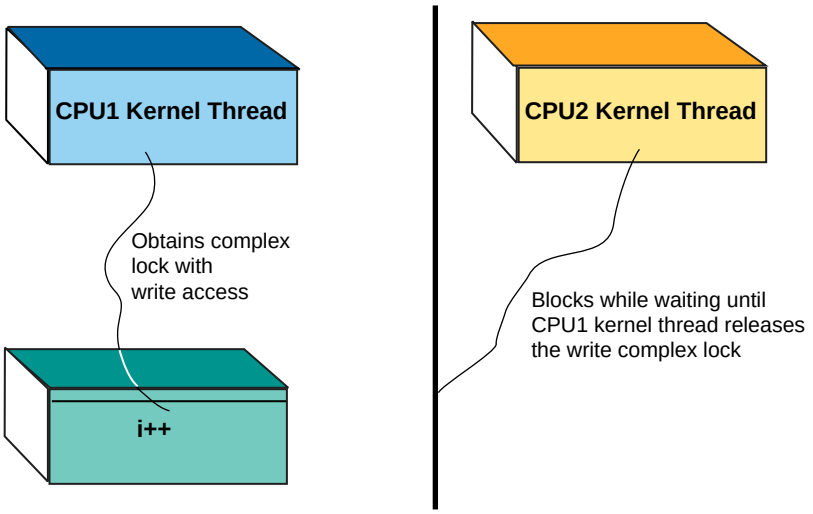
The routines that implement complex locks synchronize access to kernel data between multiple kernel threads. The following describes the characteristics of complex locks:

- Multiple reader access
- Thread blocking (sleeping) if the write lock is asserted

Figure 6–3 shows that complex locks are not spin locks, but blocking (sleeping) locks. That is, the routines that implement the complex lock block (sleep) until the lock is released. Therefore, unlike for simple locks, do not use complex locks to synchronize with interrupt service routines. Because of the blocking characteristic of complex locks, they are active on both single and multiple CPUs to serialize access to data between kernel threads.

As the figure shows, the CPU1 kernel thread asserts a complex lock with write access on resource `i`. The CPU2 kernel thread also asserts a complex lock with write access on resource `i`. Because the CPU1 kernel thread asserts the write complex lock on resource `i` first, the CPU2 kernel thread blocks, waiting until the CPU1 kernel thread unlocks (frees) the complex write lock.

Figure 6–3: Complex Locks Are Blocking Locks



ZK-0956U-AI

Like simple locks, complex locks present tradeoffs in performance and real-time preemption latency that you should understand before you use them. However, kernel modules must often use complex locks. For example, kernel modules must use complex locks when there are blocking conditions in the code block. On the other hand, you must not take a complex lock while holding a simple lock or when using the `timeout` routine. Section 6.4 provides guidelines to help you choose between simple locks and complex locks.

Table 6–2 lists the data structure and routines for complex locks. Chapter 8 discusses how to use the data structure and routines to implement complex locks in a kernel module.

Table 6–2: Data Structure and Routines Associated with Complex Locks

Structure/Routines	Description
`lock`	Contains complex lock–specific information.
`lock_done`	Releases a complex lock.
`lock_init`	Initializes a complex lock.
`lock_read`	Asserts a complex lock with read-only access.
`lock_terminate`	Terminates, using a complex lock.
`lock_try_read`	Tries to assert a complex lock with read-only access.
`lock_try_write`	Tries to assert a complex lock with write access.
`lock_write`	Asserts a complex lock with write access.

6.4 Choosing a Locking Method

You can make your kernel modules SMP-safe by implementing a simple or complex locking method.

This section provides guidelines to help you choose the appropriate locking method (simple or complex). In choosing a locking method, consider the following SMP characteristics:

- Who has access to a particular resource
- Prevention of access to the resource while a kernel thread sleeps
- Length of time the lock is held
- Execution speed
- Size of code blocks

The following sections discuss each of these characteristics. See Section 6.4.6 for a summary comparison table of the locking methods that you can use to determine which items to lock in your kernel modules.

6.4.1 Who Has Access to a Particular Resource

To choose the appropriate lock method, you must understand the entity that has access to a particular resource. Possible entities that can access a resource are kernel threads, interrupt service routines, and exceptions. If you need a lock for resources that multiple kernel threads access, use simple or complex locks. Use a combination of `spl` routines and simple locks to lock resources that kernel threads and interrupt service routines access.

For exceptions, use complex locks if the exception involves blocking conditions. If the exception does not involve blocking conditions, use simple locks.

6.4.2 Prevention of Access to a Resource While a Kernel Thread Sleeps

You must determine if it is necessary to prevent access to the resource while a kernel thread blocks (sleeps). One example is waiting for disk I/O to a buffer. If you need a lock to prevent access to the resource while a kernel thread blocks (sleeps) and there are no blocking conditions, use simple or complex locks. Otherwise, if there are blocking conditions, use complex locks.

6.4.3 Length of Time the Lock Is Held

You must estimate the length of time that the lock is held to determine the appropriate lock method. In general, use simple locks when the entity accesses are bounded and small. One example of a bounded and small access is some entity that accesses a system time variable. Use complex locks when the entity accesses might take a long time or a variable amount of time. One example of a variable amount of time is some entity scanning linked lists.

6.4.4 Execution Speed

You must account for execution speed in choosing the appropriate lock method. The following factors influence execution speed:

- The way complex locks work

  Complex locks are slightly more than twice as expensive (in terms of execution speed) as simple locks because complex locks use the simple lock routines to implement the lock. Therefore, it takes two lock and unlock pairs to protect a resource or code block with a complex lock as opposed to one pair for the simple lock.

- Memory space used

  Complex locks use more memory space than simple locks because the complex lock structure, `lock`, contains a pointer to a simple lock structure in addition to other data to implement the complex lock.

- Busy wait time

  Busy wait time is the amount of CPU time that is expended on waiting for a simple lock to become free. If the kernel module initiates a simple lock on a resource and the code block is long (or there are numerous interrupts), a lot of CPU time could be wasted waiting for the simple lock to become free. If this is the case, use complex locks to allow the current kernel thread to block (sleep) on the busy resource. This action allows the CPU to execute a different kernel thread.

- Real-time preemption

  Real-time preemption cannot occur when a simple lock is held. The use of complex locks (which can block) improves the performance that is associated with real-time preemption.

6.4.5 Size of Code Blocks

In general, use complex locks for resources that are contained in long code blocks. Also, use complex locks in cases where the resource must be prevented from changing when a kernel thread blocks (sleeps).

Use simple locks for resources that are contained in short, nonblocking code blocks or when synchronizing with interrupt service routines.

6.4.6 Summary of Locking Methods

Table 6–3 summarizes the SMP characteristics for choosing the appropriate lock method to make your kernel module SMP safe. The first column of the table presents an SMP characteristic and the second and third columns present the lock methods.

The following list describes the possible entities that can appear in the second and third columns:

- Yes — The lock method is suitable for the characteristic.
- No — The lock method is not suitable for the characteristic.
- Better — This lock method is the most suitable for the characteristic.
- Worse — This lock method is not the most suitable for the characteristic.

(The numbers before each Characteristic item appear for easy reference in later descriptions.)

Table 6–3: SMP Characteristics for Locking

Characteristic	Simple Lock	Complex Lock
1. Kernel threads can access this resource.	Yes	Yes
2. Interrupt service routines can access this resource.	Yes	No
3. Exceptions can access this resource.	Yes	Yes
4. You need to prevent access to this resource while a kernel thread blocks and there are no blocking conditions.	Yes	Yes
5. You need to prevent access to this resource while a kernel thread blocks and there are blocking conditions.	No	Yes
6. You need to protect resources between kernel threads and interrupt service routines.	Yes	No
7. You need to have maximum execution speed for this kernel module.	Yes	No
8. The module references and updates this resource in long code blocks (implying that the length of time that the lock is held on this resource is not bounded and long).	Worse	Better
9. The module references and updates this resource in short nonblocking code blocks (implying that the length of time that the lock is held on this resource is bounded and short).	Better	Worse
10. You need to minimize memory usage by the lock-specific data structures.	Yes	No
11. You need to synchronize with interrupt service routines.	Yes	No
12. The module can afford busy wait time.	Yes	No
13. The module implements real-time preemption.	Worse	Better

Use the following steps to analyze your kernel module to determine which items to lock and which locking method to choose:

1. Identify all of the resources in your kernel module that you could potentially lock. Section 6.5 discusses some of these resources.
2. Identify all of the code blocks in your kernel module that manipulate the resource.
3. Determine which locking method is appropriate. Use Table 6–3 as a guide to help you choose the locking method. Section 6.5.5 describes how to use this table for choosing a locking method for the example device register offset definition resources.
4. Determine the granularity of the lock. Section 6.5.5 describes how to determine the granularity of the locks for the example device register offset definitions.

6.5 Choosing the Resources to Lock in the Module

Section 6.4 presents the SMP characteristics to consider when you choose a locking method. You need to analyze each section of the kernel module (in device drivers, for example, the open and close device section, the read and write device section, and so forth) and apply those SMP characteristics to the following resource categories:

- Read-only resources (Section 6.5.1)
- Device control status register (CSR) addresses (Section 6.5.2)
- Module-specific global resources (Section 6.5.3)
- System-specific global resources (Section 6.5.4)

The following sections discuss each of these categories. See Section 6.5.5 for an example that walks you through the steps for analyzing a kernel module to determine which resources to lock.

6.5.1 Read-Only Resources

Analyze each section of your kernel module to determine if the access to a resource is read only. In this case, resource refers to module and system data that is stored in global variables or data structure fields. You do not need to lock resources that are read only because there is no way to corrupt the data in a read-only operation.

6.5.2 Device Control Status Register Addresses

Analyze each section of your kernel module to determine accesses to a device's control status register (CSR) addresses. Many kernel modules that

are based on the UNIX operating system use the direct method; that is, they access a device's CSR addresses directly through a device register structure. This method involves declaring a device register structure that describes the device's characteristics, which include a device's control status register. After declaring the device register structure, the kernel module accesses the device's CSR addresses through the field that maps to it.

Some CPU architectures do not allow you to access the device CSR addresses directly. Make kernel modules that need to operate on these types of CPUs use the indirect method. In fact, kernel modules that operate on Alpha systems must use the indirect method. Therefore, the discussion of locking a device's CSR addresses focuses on the indirect method.

The indirect method involves defining device register offset definitions (instead of a device register structure) that describe the device's characteristics, which include a device's control status register. The method also includes the use of the following categories of routines:

- CSR I/O access routines

  `read_io_port` — Reads data from a device register
  `write_io_port` — Writes data to a device register

- I/O copy routines

  `io_copyin`— Copies data from bus address space to system memory
  `io_copyio`— Copies data from bus address space to bus address space
  `io_copyout` — Copies data from system memory to bus address space

Using these routines makes your kernel module more portable across different bus architectures, different CPU architectures, and different CPU types within the same architecture. For examples of how to use these routines when writing device drivers, see *Writing Device Drivers*. The following example shows the device register offset definitions that some `xx` kernel module defines for some `XX` device:

```
.
.
.
#define XX_ADDER 0x0 /* 32-bit read/write DMA address register */
#define XX_DATA  0x4 /* 32-bit read/write data register */
#define XX_CSR   0x8 /* 16-bit read/write CSR/LED register */
#define XX_TEST  0xc /* Go bit register. Write sets. Read clears */
.
.
.
```

6.5.3 Module-Specific Global Resources

Analyze the declarations and definitions sections of your kernel module to identify the following global resources:

- Module-specific global variables
- Module-specific data structures

Module-specific global variables can store a variety of information, including flag values that control execution of code blocks and status information. The following example shows the declaration and initialization of some typical module-specific global variables. Use this example to help you locate similar module-specific global variables in your kernel module.

```
.
.
.
int num_xx = 0;
.
.
.

int xx_is_dynamic = 0;
.
.
.
```

Module-specific data structures contain fields that can store such information as whether a device is attached, whether it is opened, the read/write mode, and so forth. The following example shows the declaration and initialization of some typical module-specific data structures. Use this example to help you locate similar module-specific data structures in your kernel modules.

```
.
.
.
struct  driver xxdriver = {
.
.
.
};
.
.
.
cfg_subsys_attr_t xx_attributes[] = {
.
.
.
};
.
.
.
};
.
.
.
struct xx_kern_str {
.
.
.
} xx_kern_str[NXX];
.
.
.
struct cdevsw xx_cdevsw_entry = {
.
.
.
```

```
};
```

After you identify the module-specific global variables and module-specific data structures, locate the code blocks in which the kernel module references them. Use Table 6–3 to determine which locking method is appropriate. Also, determine the granularity of the lock.

6.5.4 System-Specific Global Resources

Analyze the declarations and definitions sections of your kernel module to identify the following global resources:

- System-specific global variables
- System-specific data structures

System-specific variables include the global variables *hz*, *cpu*, and *lbolt*. The following example shows the declaration of one system-specific global variable:

```
.
.
.
extern int hz;
.
.
.
```

System-specific data structures include `controller`, `buf`, and `ihandler_t`. The following example shows the declaration of some system-specific data structures:

```
.
.
.
struct controller *info[NXX];
.
.
.

struct buf cbbuf[NCB];
.
.
.
```

After you identify the system-specific global variables and system-specific data structures, locate the code blocks in which the module references them. Use Table 6–3 to determine which locking method is appropriate. Also, determine the granularity of the lock.

Note

To lock `buf` structure resources, use the `BUF_LOCK` and `BUF_UNLOCK` routines instead of the simple and complex lock routines. For descriptions of these routines, see `BUF_LOCK`(9) and `BUF_UNLOCK`(9).

6.5.5 How to Determine the Resources to Lock

Use the following steps to determine which resources to lock in your kernel modules:

1. Identify all resources that you might lock.
2. Identify all of the code blocks in the kernel module that manipulate each resource.
3. Determine which locking method is appropriate.
4. Determine the granularity of the lock.

The following sections provide an example that walks you through an analysis of which resources to lock for the `xx` module.

6.5.5.1 Step 1: Identify All Resources That You Might Lock

Table 6–4 summarizes the resources that you might lock in your kernel module according to the following categories:

- Device control status register (CSR) addresses
- Module-specific global variables
- Module-specific data structures
- System-specific global variables
- System-specific global data structures

Table 6–4: Kernel Module Resources for Locking

Category	Associated Resources
Device control status register (CSR) addresses.	N/A
Module-specific global variables.	Variables that store flag values to control execution of code blocks. Variables that store status information.
Module-specific global data structures.	`dsent`, `cfg_subsys_attr_t`, `driver`, and the kernel module's `kern_str` structure.

Table 6–4: Kernel Module Resources for Locking (cont.)

Category	Associated Resources
System-specific global variables	*cpu*, *hz*, *lbolt*, and *page_size*.
System-specific global data structures	`controller` and `buf`.

One resource that the `xx` module must lock is the device CSR addresses. This module also needs to lock the *hz* global variable. The example analysis focuses on the following device register offset definitions for the `xx` module:

```
.
.
#define XX_ADDER 0x0 /* 32-bit read/write DMA address register */
#define XX_DATA  0x4 /* 32-bit read/write data register */
#define XX_CSR   0x8 /* 16-bit read/write CSR/LED register */
#define XX_TEST  0xc /* Go bit register. Write sets. Read clears */
.
.
```

6.5.5.2 Step 2: Identify All of the Code Blocks in the Module That Manipulate the Resource

Identify all of the code blocks that manipulate the resource. If the code block accesses the resource read only, you might not need to lock the resources that it references. However, if the code block writes to the resource, you need to lock the resource by calling the simple or complex lock routines.

The `xx` module accesses the device register offset definition resources in the open and close device section and the read and write device section.

6.5.5.3 Step 3: Determine Which Locking Method Is Appropriate

Table 6–5 shows how to analyze the locking method that is most suitable for the device register offset definitions for some `xx` module. (The numbers before each Characteristic item appear for easy reference in later descriptions.)

Table 6–5: Locking Device Register Offset Definitions

Characteristic	Applies to This Module	Simple Lock	Complex Lock
1. Kernel threads can access this resource.	Yes	Yes	Yes
2. Interrupt service routines can access this resource.	No	N/A	N/A
3. Exceptions can access this resource.	No	N/A	N/A
4. You need to prevent access to this resource while a kernel thread blocks and there are no blocking conditions.	Yes	Yes	Yes
5. You need to prevent access to this resource while a kernel thread blocks and there are blocking conditions.	No	N/A	N/A
6. You need to protect resources between kernel threads and interrupt service routines.	Yes	Yes	No
7. You need to have maximum execution speed for this kernel module.	Yes	Yes	No
8. The module references and updates this resource in long code blocks (implying that the length of time that the lock is held on this resource is not bounded and long).	No	N/A	N/A
9. The module references and updates this resource in short nonblocking code blocks (implying that the length of time that the lock is held on this resource is bounded and short).	Yes	Better	Worse
10. You need to minimize memory usage by the lock-specific data structures.	Yes	Yes	No
11. You need to synchronize with interrupt service routines.	No	N/A	N/A
12. The module can afford busy wait time.	Yes	Yes	No
13. The module implements real-time preemption.	No	N/A	N/A

The locking analysis table for the device register offset definitions shows the following:

- Seven of the SMP characteristics (numbers 1, 4, 6, 7, 9, 10, and 12) apply to the xx module.
- Simple and complex locks are suitable for SMP characteristics 1 and 4.
- Simple locks are better suited than complex locks for SMP characteristic 9.

- Simple locks (not complex locks) are suitable for SMP characteristics 6, 7, 10, and 12.

Based on the previous analysis, the xx module uses the simple lock method.

6.5.5.4 Step 4: Determine the Granularity of the Lock

After you choose the appropriate locking method for the resource, determine the granularity of the lock. For example, in the case of the device register offset resource, you can determine the granularity by answering the following questions:

1. Is a simple lock needed for each device register offset definition?
2. Is one simple lock needed for all of the device register offset definitions?

Table 6–5 indicates that minimizing memory usage is important to the xx module; therefore, creating one simple lock for all of the device register offset definitions saves the most memory. The following code fragment shows how to declare a simple lock for all of the device register offset definitions:

```
.
.
#include <kern/lock.h>
.
.

decl_simple_lock_data( , slk_xxdevoffset)
.
.
```

If the preservation of memory were not important to the xx module, declaring a simple lock for each device register offset definition might be more appropriate. The following code fragment shows how to declare a simple lock structure for each of the example device register offset definitions:

```
.
.
#include <kern/lock.h>
.
.

decl_simple_lock_data( , slk_xxaddr)
decl_simple_lock_data( , slk_xxdata)
decl_simple_lock_data( , slk_xxcsr)
decl_simple_lock_data( , slk_xxtest)
.
.
```

After declaring a simple lock structure for an associated resource, you must initialize it (only once) by calling `simple_lock_init`. Use the simple lock routines in code blocks that access the resource. Chapter 7 discusses the simple lock–related routines.

7

Simple Lock Routines

After you decide that the simple lock method is the appropriate method for locking specific resources, you use the simple lock routines to accomplish the locking. To use simple locks in a kernel module, perform the following tasks:

- Declare a simple lock data structure (Section 7.1)
- Initialize a simple lock (Section 7.2)
- Assert exclusive access on a resource (Section 7.3)
- Release a previously asserted simple lock (Section 7.4)
- Try to obtain a simple lock (Section 7.5)
- Terminate a simple lock (Section 7.6)
- Use the `spl` routines with simple locks (Section 7.7)

To illustrate the use of these routines, the chapter uses code from an example kernel module called `xx` that operates on some `XX` device. This example module locks a `kern_str` structure resource called `xx_kern_str`.

7.1 Declaring a Simple Lock Data Structure

Before you use a simple lock, declare a simple lock data structure for the resource that you want to lock by using the `decl_simple_lock_data` macro. The following code fragment shows a call to `decl_simple_lock_data` in the `xx` kernel module:

```
.
.
.
#include <kern/lock.h> 1
.
.
.

struct xx_kern_str {
  int sc_openf; /* Open flag */
  int sc_count; /* Count of characters written to device */
  decl_simple_lock_data( , lk_xx_kern_str); /* SMP lock for xx_kern_str */
}xx_kern_str[NNONE]; 2
.
.
.
```

1 Includes the header file `/usr/sys/include/kern/lock.h`. The `lock.h` file defines the simple spin lock and complex lock structures that the kernel modules use for synchronization on single-processor and multiprocessor systems.

2 Declares an array of kern_str structures and calls it xx_kern_str. The xx module uses the decl_simple_lock_data macro to declare a simple lock structure as a field of the xx_kern_str structure.

The decl_simple_lock_data macro declares a simple lock structure, slock, of the specified *name*. You declare a simple lock structure to protect kernel module data structures and device register access. You use decl_simple_lock_data to declare a simple lock structure and then pass it to the following simple lock-specific routines: simple_lock_init, simple_lock, simple_lock_try, simple_unlock, and simple_lock_terminate.

The decl_simple_lock_data macro can take two arguments:

- The first argument (not passed in this call) specifies the class of the declaration. For example, you pass the extern keyword if you want to declare the simple lock structure as an external structure. This argument is specified in this call if lk_xx_kern_str is declared in another program module.
- The second argument specifies the name that you want the decl_simple_lock_data routine to assign to the declaration of the simple lock structure. In this call to the routine, the name for the simple lock structure is lk_xx_kern_str.

Do not follow an invocation to the decl_simple_lock_data macro with a semicolon.

You can also declare a simple lock structure by using the typedef simple_lock_data_t, as in the following example:

```
.
.
struct xx_kern_str {
  int sc_openf; /* Open flag */
  int sc_count; /* Count of characters written to device */
  simple_lock_data_t lk_xx_kern_str; /* SMP lock for xx_kern_str */
}xx_kern_str[NNONE];  1
```

1 Declares an array of kern_str structures and calls it xx_kern_str. The xx module declares a simple lock structure as a field of the xx_kern_str structure to protect the integrity of the data that is stored in the sc_openf and sc_count fields. A kernel module's kern_str structure is one resource that often requires protection in an SMP environment because kernel module routines use it to share data. More than one kernel thread might need to access the fields of an xx_kern_str structure.

7.2 Initializing a Simple Lock

After declaring the simple lock data structure, you initialize it by calling the simple_lock_init routine. The following code fragment shows a call

to simple_lock_init by the xx kernel module's xxcattach routine. The xxcattach routine performs the tasks that are necessary to establish communication with the actual device. One of these tasks is to initialize any global data structures. Therefore, the xxcattach routine initializes the simple lock structure lk_xx_kern_str.

The code fragment also shows the declaration of the simple lock structure in the xx_kern_str structure.

```
.
.
.
#include <kern/lock.h> 1
.
.
.

struct xx_kern_str {
  int sc_openf; /* Open flag */
  int sc_count; /* Count of characters written to device */
  simple_lock_data_t lk_xx_kern_str; /* SMP lock for xx_kern_str */
}xx_kern_str[NNONE]; 2
.
.
.

xxcattach(struct controller *ctlr)
{
  register struct xx_kern_str *sc = &xx_kern_str[ctlr->ctlr_num];

  /* Tasks to perform controller-specific initialization */
.
.
.

simple_lock_init(&sc->lk_xx_kern_str); 3
.
.
.

 /* Perform any other controller-specific initialization tasks */
}
```

1 Includes the /usr/sys/include/kern/lock.h header file. The lock.h file defines the simple spin lock and complex lock structures that the kernel modules use for synchronization on single-processor and multiprocessor systems.

2 Declares an array of kern_str structures and calls it xx_kern_str. The xx kernel module declares a simple lock structure as a field of the xx_kern_str structure to protect the integrity of the data that is stored in the sc_openf and sc_count fields. A kernel module's kern_str structure is one resource that often requires protection in an SMP environment because kernel module routines use it to share data. More than one kernel thread might need to access the fields of an xx_kern_str structure.

3 Calls the simple_lock_init routine to initialize the simple lock structure called lk_xx_kern_str.

The simple_lock_init routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure

by using the decl_simple_lock_data macro. In this call, the xxcattach routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer. You need to initialize the simple lock structure only once.

7.3 Asserting Exclusive Access on a Resource

After you declare and initialize the simple lock data structure, you can assert exclusive access by calling the simple_lock routine. The following code fragment shows a call to simple_lock by the xx kernel module's xxopen routine.

The xxopen routine is called as the result of an open system call.

The xxopen routine performs the following tasks:

- Ensures that the open is unique
- Marks the device as open
- Returns the value 0 (zero) to the open system call to indicate success

The code fragment also shows the declaration of the simple lock structure in the xx_kern_str structure and the initialization of the simple lock structure by thel module's *xx*cattach routine. See Section 7.2 for explanations of these tasks.

```
.
.
#include <kern/lock.h>
.
.

struct xx_kern_str {
  int sc_openf; /* Open flag */
  int sc_count; /* Count of characters written to device */
  simple_lock_data_t lk_xx_kern_str; /* SMP lock for xx_kern_str */
}xx_kern_str[NXX];
.
.

xxcattach(struct controller *ctlr)
{
  register struct xx_kern_str *sc = &xx_kern_str[ctlr->ctlr_num];

  /* Tasks to perform controller-specific initialization */
.
.

simple_lock_init(&sc->lk_xx_kern_str);
.
.

}
.
.

xxopen(dev, flag, format)
  dev_t dev;
```

```
  int flag;
  int format;

  register int unit = minor(dev);
  struct controller *ctlr = xxinfo[unit];
  struct xx_kern_str *sc = &xx_kern_str[unit];

  if(unit >= NXX)
        return ENODEV; 1
  simple_lock(&sc->lk_xx_kern_str); 2
  if (sc->sc_openf == DN_OPEN)
  {
.
.
.
}
```

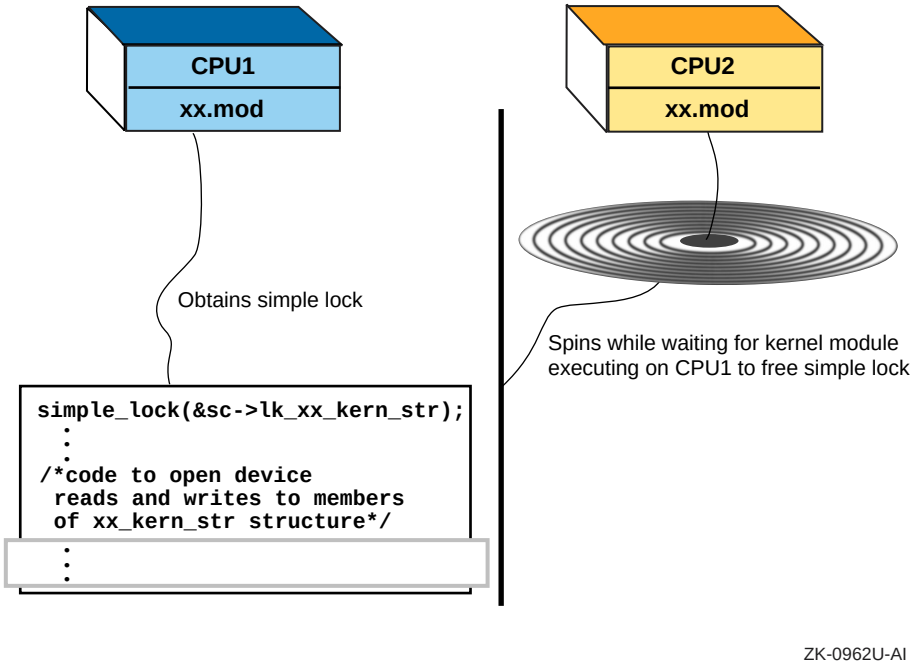
1 If the number of device units on the system is greater than NXX, returns the error code ENODEV, which indicates that no such device exists on the system. This example test ensures that a valid device exists.

2 Calls the simple_lock routine to assert an exclusive access on the following code block.

The simple_lock routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the decl_simple_lock_data macro. In this call, the xxopen routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer.

Figure 7–1 shows what happens when two instances of the xx kernel module execute on two CPUs. As the figure shows, the kernel thread from CPU1 obtains the simple lock on the code block that follows item 2 in the code fragment before the kernel thread from CPU2. The reason for locking this code block is to prevent data corruption of any future writes to the xx_kern_str structure. The CPU2 kernel thread spins while it waits for the CPU1 kernel thread to free the simple lock.

Figure 7–1: Two Instances of the xx Module Asserting an Exclusive Lock



7.4 Releasing a Previously Asserted Simple Lock

After you assert a simple lock (with exclusive access), you must release the lock by calling the `simple_unlock` routine. The following code fragment shows calls to `simple_unlock` by the `xx` kernel module's `xxopen` routine.

The `xxopen` routine is called as the result of an `open` system call.

The `xxopen` routine performs the following tasks:

- Ensures that the open is unique
- Marks the device as open
- Returns the value 0 (zero) to the `open` system call to indicate success

The code fragment also shows the declaration of the simple lock structure in the `xx_kern_str` structure and the initialization of the simple lock structure by the kernel module's *xx*`cattach` routine.

```
.
.
#include <kern/lock.h>
.
.

struct xx_kern_str {
  int sc_openf; /* Open flag */
  int sc_count; /* Count of characters written to device */
  simple_lock_data_t lk_xx_kern_str; /* SMP lock for xx_kern_str */
}xx_kern_str[NXX];
.
.

xxcattach(struct controller *ctlr)
{
  register struct xx_kern_str *sc = &xx_kern_str[ctlr->ctlr_num];

  /* Tasks to perform controller-specific initialization */
.
.

simple_lock_init(&sc->lk_xx_kern_str);
.
.

}
.
.

xxopen(dev, flag, format)
  dev_t dev;
  int flag;
  int format;

  register int unit = minor(dev);
  struct controller *ctlr = xxinfo[unit];
  struct xx_kern_str *sc = &xx_kern_str[unit];

  if(unit >= NXX)
        return ENODEV; 1
  simple_lock(&sc->lk_xx_kern_str); 2
  if (sc->sc_openf == DN_OPEN) 3
  {
                simple_unlock(&sc->lk_xx_kern_str);
                return (EBUSY);
        }
        if ((ctlr !=0) && (ctlr->alive & ALV_ALIVE)) 4
        {
                sc->sc_openf = DN_OPEN;
                simple_unlock(&sc->lk_xx_kern_str);
                return(0);
        }
        else 5
        {
                simple_unlock(&sc->lk_xx_kern_str);
                return(ENXIO);
        }
}
.
.
```

1 If the number of device units on the system is greater than NXX, returns the error code ENODEV, which indicates that no such device exists on the system. This example test ensures that a valid device exists.

2 Calls the simple_lock routine to assert an exclusive access on the following code block.

The simple_lock routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the decl_simple_lock_data macro. In this call, the xxopen routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer.

3 If the sc_openf field of the sc pointer is equal to DN_OPEN, calls the simple_unlock routine and returns the error code EBUSY, which indicates that the NONE device has already been opened. This example test ensures that only one unit of the kernel module can be opened at a time. This type of open is referred to as an **exclusive access** open.

The simple_unlock routine releases a simple lock for the resource that is associated with the specified simple lock structure pointer. This simple lock was previously asserted by calling the simple_lock or simple_lock_try routine. In this call, the locked resource is referenced in the code block beginning with item 3.

The simple_unlock routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the decl_simple_lock_data macro. In this call, the xxopen routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer.

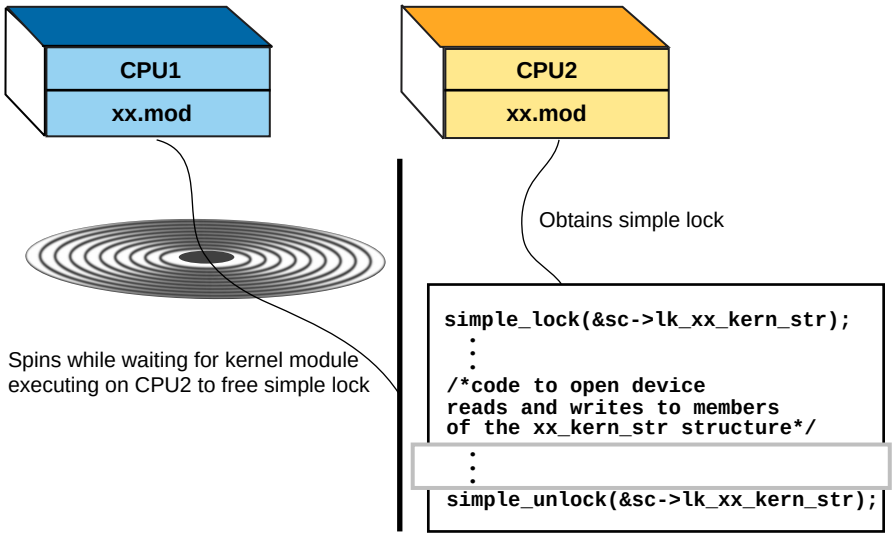
4 If the ctlr pointer is not equal to 0 (zero) and the alive field of ctlr has the ALV_ALIVE bit set, then the device exists. If this is the case, the xxopen routine sets the sc_openf field of the sc pointer to the open bit DN_OPEN, calls simple_unlock to free the lock, and returns the value 0 (zero) to indicate a successful open.

5 If the device does not exist, xxopen calls simple_unlock to free the lock and returns the error code ENXIO, which indicates that the device does not exist.

Figure 7–2 shows what happens when one instance of the xx kernel module releases a previously asserted exclusive lock on the code block that opens the device. In Figure 7–1, the CPU1 kernel thread obtained the simple lock on the code block that opens the device. The CPU2 kernel thread spun while it waits for the simple lock to be freed. After CPU 1 released the simple lock, CPU2 obtained the lock. In Figure 7–2, the CPU1 kernel thread makes another attempt to lock the code block

that opens the device. This time it spins until the CPU2 kernel thread releases the simple lock.

Figure 7–2: One Instance of the xx Module Releasing an Exclusive Lock



7.5 Trying to Obtain a Simple Lock

In addition to explicitly asserting a simple lock, you can also try to assert the simple lock by calling the `simple_lock_try` routine. The main difference between `simple_lock` and `simple_lock_try` is that `simple_lock_try` returns immediately if the resource is already locked, while `simple_lock` spins until the lock has been obtained. Therefore, call `simple_lock_try` when you need a simple lock but the code cannot spin until the lock is obtained.

The following code fragment shows a call to `simple_lock_try` by the `xx` kernel module's `xxopen` routine.

The `xxopen` routine is called as the result of an `open` system call.

The `xxopen` routine performs the following tasks:

- Ensures that the open is unique
- Marks the device as open
- Returns the value 0 (zero) to the `open` system call to indicate success

The code fragment also shows the declaration of the simple lock structure in the xx_kern_str structure and the initialization of the simple lock structure by the kernel module's *xx*cattach routine.

```
.
.
#include <kern/lock.h>
.
.

struct xx_kern_str {
  int sc_openf; /* Open flag */
  int sc_count; /* Count of characters written to device */
  simple_lock_data_t lk_xx_kern_str; /* SMP lock for xx_kern_str */
}xx_kern_str[NXX];
.
.

xxcattach(struct controller *ctlr)
{
  register struct xx_kern_str *sc = &xx_kern_str[ctlr->ctlr_num];

  /* Tasks to perform controller-specific initialization */
.
.

simple_lock_init(&sc->lk_xx_kern_str);
.
.

}
.
.

xxopen(dev, flag, format)
  dev_t dev;
  int flag;
  int format;

  register int unit = minor(dev);
  struct controller *ctlr = xxinfo[unit];
  struct xx_kern_str *sc = &xx_kern_str[unit];
  boolean_t try_ret_val; [1]

  if(unit >= NXX)
        return ENODEV; [2]
  try_ret_val = simple_lock_try(&sc->lk_xx_kern_str); [3]
  if (try_ret_val == TRUE)  [4]
  {
    if (sc->sc_openf == DN_OPEN)
.
.

  else
/* Perform some other tasks if simple_lock_try fails *
 * to assert an exclusive access                     */
.
.

}
```

[1] Declares a variable to store the return value from the simple_lock_try routine.

The simple_lock_try routine returns one of the following values:

TRUE	The simple_lock_try routine successfully asserted the simple lock.
FALSE	The simple_lock_try routine failed to assert the simple lock.

2 If the number of device units on the system is greater than NXX, returns the error code ENODEV, which indicates that no such device exists on the system. This example test ensures that a valid device exists.

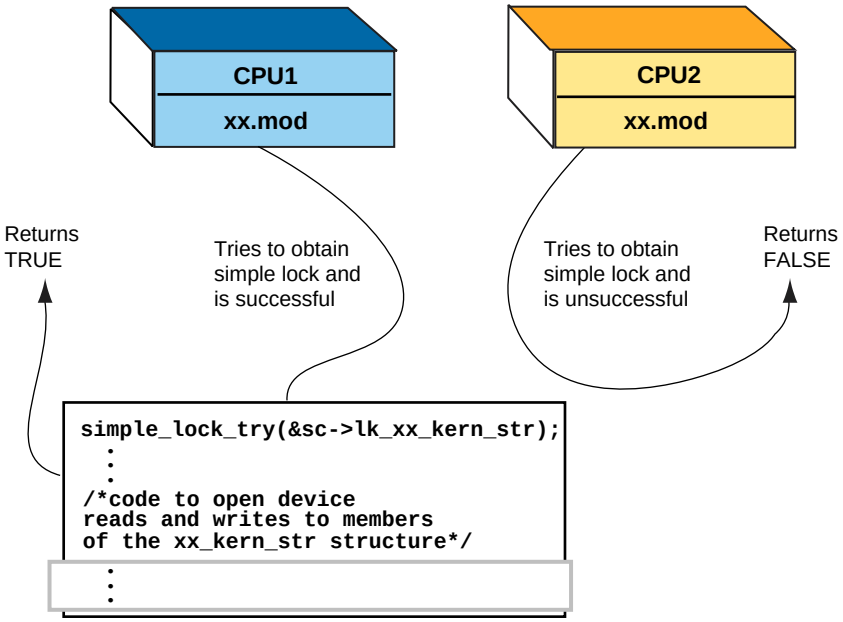
3 Calls the simple_lock_try routine to try to assert an exclusive access on the following code block.

The simple_lock_try routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the decl_simple_lock_data macro. In this call, the xxopen routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer.

4 If the return from simple_lock_try is TRUE, examines the sc_openf field to determine whether this is a unique open. Otherwise, if the return from simple_lock_try is FALSE, performs some other tasks.

Figure 7–3 shows what happens when two instances of the xx kernel module try to assert an exclusive lock on the code block that opens the device. As the figure shows, the CPU1 and CPU2 kernel threads try to assert an exclusive lock on the code block that opens the device. In this case, the CPU1 kernel thread successfully obtains the lock. To indicate this success, simple_lock_try returns the value TRUE. At the same time, the CPU2 kernel thread fails to obtain the lock and simple_lock_try immediately returns the value FALSE to indicate this.

Figure 7–3: The xx Module Trying to Assert an Exclusive Lock



ZK-0964U-AI

7.6 Terminating a Simple Lock

After you unlock a simple lock (with exclusive access) and know that you are finished using the lock for this resource, you can terminate the lock by calling the `simple_lock_terminate` routine. Typically, you terminate any locks in the kernel module's controller (or device) `unattach` routine. These routines are associated with loadable modules (for example, drivers). One task that is associated with a controller or device `unattach` routine is to terminate any locks that are initialized in the kernel module's `unattach` routine.

The following code fragment shows a call to `simple_lock_terminate` by the `xx` kernel module's `xx_ctlr_unattach` routine. The code fragment also shows the declaration of the simple lock structure in the `xx_kern_str` structure, the initialization of the simple lock structure by the kernel module's `xxcattach` routine, and the unlocking of the simple lock structure by the module's `xxopen` routine.

```
.
.
.
#include <kern/lock.h> 1
.
.
.

struct xx_kern_str {
  int sc_openf; /* Open flag */
```

```
  int sc_count; /* Count of characters written to device */
  simple_lock_data_t lk_xx_kern_str; /* SMP lock for xx_kern_str */
}xx_kern_str[NXX]; 2
.
.
.

xxcattach(struct controller *ctlr)
{
  register struct xx_kern_str *sc = &xx_kern_str[ctlr->ctlr_num];

  /* Tasks to perform controller-specific initialization */
.
.
.

simple_lock_init(&sc->lk_xx_kern_str); 3
.
.
.

}
.
.
.

xxopen(dev, flag, format)
  dev_t dev;
  int flag;
  int format;

  register int unit = minor(dev);
  struct controller *ctlr = xxinfo[unit];
  struct xx_kern_str *sc = &xx_kern_str[unit];

  if(unit >= NXX)
        return ENODEV; 4
  simple_lock(&sc->lk_xx_kern_str); 5
  if (sc->sc_openf == DN_OPEN) 6
  {
                simple_unlock(&sc->lk_xx_kern_str);
                return (EBUSY);
        }
        if ((ctlr !=0) && (ctlr->alive & ALV_ALIVE)) 7
        {
                sc->sc_openf = DN_OPEN;
                simple_unlock(&sc->lk_xx_kern_str);
                return(0);
        }
        else 8
        {
                simple_unlock(&sc->lk_xx_kern_str);
                return(ENXIO);
        }
}
.
.
.

xx_ctlr_unattach(bus, ctlr)
  struct bus *bus;
  struct controller *ctlr;
{
        register int unit = ctlr->ctlr_num;

        if ((unit > num_xx) || (unit < 0) {
                return(1);
        }

        if (xx_is_dynamic == 0) {
```

```
                return(1);
}
/* Performs controller unattach tasks */
.
.
.
                 simple_lock_terminate(&sc->lk_xx_kern_str); 9
```

1 Includes the /usr/sys/include/kern/lock.h header file. The lock.h file defines the simple spin lock and complex lock structures that the kernel modules use for synchronization on single-processor and multiprocessor systems.

2 Declares an array of kern_str structures and calls it xx_kern_str. The xx kernel module declares a simple lock structure as a field of the xx_kern_str structure to protect the integrity of the data that is stored in the sc_openf and sc_count fields. A kernel module's kern_str structure is one resource that often requires protection in an SMP environment because kernel module routines use it to share data. More than one kernel thread might need to access the fields of an xx_kern_str structure.

3 Calls the simple_lock_init routine to initialize the simple lock structure called lk_xx_kern_str.

The simple_lock_init routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the decl_simple_lock_data macro. In this call, the xxcattach routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer. You need to initialize the simple lock structure only once. After initializing a simple lock structure, kernel modules can call simple_lock to assert exclusive access on the associated resource or simple_lock_try to attempt to assert exclusive access on the associated resource.

4 If the number of device units on the system is greater than NXX, returns the error code ENODEV, which indicates that no such device exists on the system. This example test ensures that a valid device exists.

5 Calls the simple_lock routine to assert an exclusive access on the following code block.

The simple_lock routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the decl_simple_lock_data macro. In this call, the xxopen routine passes the address of the lk_xx_kern_str field of the xx_kern_str structure pointer.

6 If the sc_openf field of the sc pointer is equal to DN_OPEN, calls the simple_unlock routine and returns the error code EBUSY, which indicates that the NONE device has already been opened. This example

test ensures that only one unit of the module can be opened at a time. This type of open is referred to as an **exclusive access** open.

The `simple_unlock` routine releases a simple lock for the resource that is associated with the specified simple lock structure pointer. This simple lock was previously asserted by calling the `simple_lock` or `simple_lock_try` routine. In this call, the locked resource is referenced in the code block beginning with item 6.

The `simple_unlock` routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the `decl_simple_lock_data` macro. In this call, the `xxopen` routine passes the address of the `lk_xx_kern_str` field of the `xx_kern_str` structure pointer.

7 If the `ctlr` pointer is not equal to 0 (zero) and the `alive` field of `ctlr` has the `ALV_ALIVE` bit set, then the device exists. If this is the case, the `xxopen` routine sets the `sc_openf` field of the `sc` pointer to the open bit `DN_OPEN`, calls `simple_unlock` to free the lock, and returns the value 0 (zero) to indicate a successful open.

8 If the device does not exist, `xxopen` calls `simple_unlock` to free the lock and returns the error code `ENXIO`, which indicates that the device does not exist.

9 Calls the `simple_lock_terminate` routine to determine that the `xx` module is permanently done using this simple lock.

The `simple_lock_terminate` routine takes one argument: a pointer to a simple lock structure. You can declare this simple lock structure by using the `decl_simple_lock_data` macro. In this call, the `xx_ctlr_unattach` routine passes the address of the `lk_xx_kern_str` field of the `xx_kern_str` structure pointer. In calling `simple_lock_terminate`, the `xx` kernel module must not reference this simple lock again.

7.7 Using the spl Routines with Simple Locks

The `spl` routines block out asynchronous events on the CPU on which the `spl` call is performed. Simple locks block out other CPUs. You need to use both the `spl` routines and the simple lock routines when you synchronize with kernel threads and interrupt service routines. The following code fragment shows calls to the `spl` and simple lock routines:

```
.
.
#include <kern/lock.h> 1
.
.

struct tty_kern_str {
```

```
.
.
.
  decl_simple_lock_data( , lk_tty_kern_str); /* SMP lock for tty_kern_str */
.
.
.
}tty_kern_str[NSOMEDEVICE]; 2
.
.
.
simple_lock_init(&sc->lk_tty_kern_str);
.
.
.
s = spltty(); 3
simple_lock(&lk_tty_kern_str); 4
.
.
.
/* Manipulate resource */
.
.
.
simple_unlock(&lk_tty_kern_str); 5
splx(s); 6
.
.
.
```

1 Includes the `/usr/sys/include/kern/lock.h` header file. The `lock.h` file defines the simple spin lock and complex lock structures that the kernel modules use for synchronization on single-processor and multiprocessor systems.

2 Declares an array of `kern_str` structures and calls it `lk_tty_kern_str`. This example module uses the `decl_simple_lock_data` macro to declare a simple lock structure as a field of the `tty_kern_str` structure.

3 Calls the `spltty` routine to mask out all tty (terminal device) interrupts.

The `spltty` routine takes no arguments.

The `spltty` routine returns an integer value that represents the CPU priority level that existed before the call. The routine masks out all tty interrupts on the CPU on which it is called.

4 Calls the `simple_lock` routine to assert a lock with exclusive access for the resource that is associated with the `slock` structure pointer, which in this example is `lk_tty_kern_str`. The routine ensures that no other kernel thread that runs on other CPUs can gain access to this resource. This contrasts with the `spl` routines, which block out kernel threads running on this CPU.

5 After manipulating the resource, calls `simple_unlock` to release the simple lock. This makes the resource available to kernel threads running on other CPUs.

6 Calls the `splx` routine to reset the CPU priority to the level specified by the value returned by `spltty`.

The `splx` routine takes one argument: a CPU priority level. This level must be a value returned by a previous call to one of the `spl` routines, in this example `spltty`. Calling `splx` releases the priority on this CPU.

8

Complex Lock Routines

After you decide that the complex lock method is the appropriate method for locking specific resources, use the complex lock routines to accomplish the locking. To use complex locks in a kernel module, perform the following tasks:

- Declare a complex lock data structure (Section 8.1)
- Initialize a complex lock (Section 8.2)
- Perform access operations on a complex lock (Section 8.3)
- Terminate a complex lock (Section 8.4)

To show the use of these routines, the chapter uses code from an example kernel module called `if_fta`, which operates on some FTA device.

8.1 Declaring a Complex Lock Data Structure

Before you use a complex lock, declare a complex lock data structure for the resource that you want to lock. The following code fragment shows how to declare a complex lock data structure for a specific field of the `fta_kern_str` structure:

```
#include <kern/lock.h> 1

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
}; 2
.
.
.

struct  fta_kern_str {
.
struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.
.

}; 3
.
.
.
```

1 Includes the /usr/sys/include/kern/lock.h header file. The lock.h file defines the simple spin lock and complex lock structures that the kernel modules use for synchronization on single-processor and multiprocessor systems.

2 Defines a cmd_buf data structure. The fta_kern_str structure declares two instances of cmd_buf. This structure describes a command queue and is a candidate for locking in a symmetric multiprocessing (SMP) environment. You must protect the integrity of the data that is stored in the command queue from multiple writes by more than one kernel thread.

3 Defines an fta_kern_str data structure. The example shows only those fields that are related to the discussion of complex locks.

In this example, the fta_kern_str structure contains the following fields:

- q_first

  Specifies a pointer to a cmd_buf data structure. This field represents the first command queue in the linked list.

- q_last

  Specifies a pointer to a cmd_buf data structure. This field represents the last command queue in the linked list.

- cmd_buf_q_lock

  Declares a lock structure called cmd_buf_q_lock. The purpose of this lock is to protect the integrity of the data that is stored in the linked list of cmd_buf data structures. The alternate name lock_data_t declares the complex lock structure. Embedding the complex lock in the fta_kern_str structure protects the cmd_buf structure for any number of instances.

8.2 Initializing a Complex Lock

After you declare the complex lock data structure, you initialize it by calling the lock_init routine. The following code fragment shows a call to lock_init by the if_fta module's ftaattach routine. The ftaattach routine performs the tasks that establish communication with the actual device. One of these tasks is to initialize any global data structures. Therefore, the ftaattach routine initializes the complex lock data structure cmd_buf_q_lock.

The code fragment also shows the include file that is associated with complex locks, definitions of the cmd_buf and fta_kern_str structures, and the declaration of the complex lock.

```
.
.
#include <kern/lock.h> [1]  .
                            .struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
}; [2]
.
.

struct  fta_kern_str {
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.

}; [3]
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.

  /* Tasks to perform controller-specific initialization */
.
.

lock_init(&sc->cmd_buf_q_lock, TRUE); [4]
.
.

 /* Perform other tasks */
}
```

[1] Includes the `/usr/sys/include/kern/lock.h` header file. The `lock.h` file defines the simple spin lock and complex lock structures that the kernel modules use for synchronization on single-processor and multiprocessor systems.

[2] Defines a `cmd_buf` data structure. The `fta_kern_str` structure declares two instances of `cmd_buf`. This structure describes a command queue and is a candidate for locking in an SMP environment. You must protect the integrity of the data that is stored in the command queue from multiple writes by more than one kernel thread.

[3] Defines an `fta_kern_str` data structure. The example shows only those fields that are related to the discussion of complex locks.

In this example, the `fta_kern_str` structure contains the following fields:

- `q_first`

Specifies a pointer to a cmd_buf data structure. This field represents the first command queue in the linked list.

- q_last

  Specifies a pointer to a cmd_buf data structure. This field represents the last command queue in the linked list.

- cmd_buf_q_lock

  Declares a lock structure called cmd_buf_q_lock. This lock protects the integrity of the data that is stored in the linked list of cmd_buf data structures. The alternate name lock_data_t declares the complex lock structure. Embedding the complex lock in the fta_kern_str structure protects the cmd_buf structure for any number of instances.

4 Calls the lock_init routine to initialize the simple lock structure called cmd_buf_q_lock.

The lock_init routine takes two arguments:

- The first argument specifies a pointer to the complex lock structure. In this call, the ftaattach routine passes the address of the cmd_buf_q_lock field of the fta_kern_str structure pointer. You need to initialize the complex lock structure only once.
- The second argument specifies a Boolean value that indicates whether to allow kernel threads to block (sleep) if the complex lock is asserted. You can pass to this argument only the value TRUE (allow kernel threads to block if the lock is asserted).

8.3 Performing Access Operations on a Complex Lock

After you declare and initialize the complex lock data structure, you can perform the following access operations on the complex lock:

- Assert a complex lock (Section 8.3.1)
- Release a previously asserted complex lock (Section 8.3.2)
- Try to assert a complex lock (Section 8.3.3)

Each of these tasks is discussed in the following sections.

8.3.1 Asserting a Complex Lock

After you declare and initialize the complex lock data structure, you can assert a complex lock with read-only access or a complex lock with write access by calling the lock_read or lock_write routine. The following sections describe how to use these routines.

8.3.1.1 Asserting a Complex Lock with Read-Only Access

The `lock_read` routine asserts a lock with read-only access for the resource that is associated with the specified `lock` structure pointer. The following code fragment shows a call to `lock_read` by the `if_fta` module's `ftaioctl` routine.

The `ftaioctl` routine is called as the result of an `ioctl` system call.

The `ftaioctl` routine performs the following tasks:

- Determines the type of request
- Executes the request
- Returns data
- Returns the value 0 (zero) to the `ioctl` system call to indicate success

The code fragment also shows the include file that is associated with complex locks, definitions of the `cmd_buf` and `fta_kern_str` structures, the declaration of the complex lock structure in the `fta_kern_str` structure, and the initialization of the complex lock structure by the kernel module's `ftaattach` routine. Section 8.2 provides descriptions of these tasks.

```
#include <kern/lock.h>
.
.
.

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
};
.
.
.

struct  fta_kern_str {
.
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.
.

};
.
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.
.

  /* Tasks to perform controller-specific initialization */
```

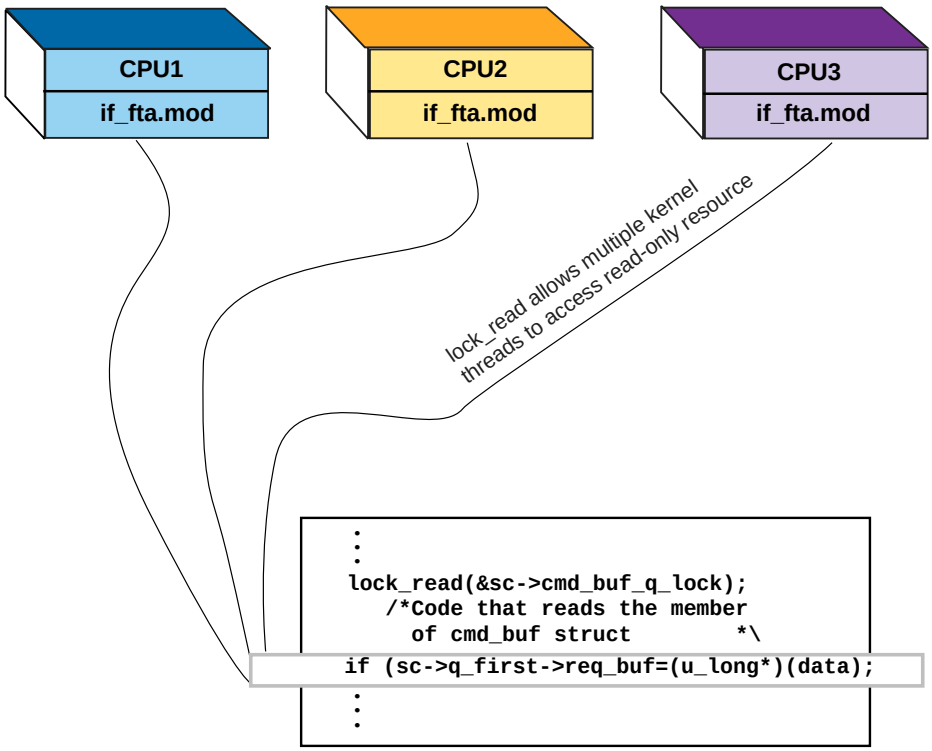
```
.
.
.
lock_init(&sc->cmd_buf_q_lock, TRUE);
.
.
.
 /* Perform other tasks */
}
.
.
.
ftaioctl(register struct ifnet *ifp,
         unsigned int cmd,
         caddr_t dataifp)
{
    struct fta_kern_str *sc = &fta_kern_str[ifp->if_unit];
.
.
.
    switch (cmd) {
      case SIOCENABLBACK: {
.
.
.
          if (ifp->if_flags & IFF_RUNNING) {[1]
              lock_read(&sc->cmd_buf_q_lock);

 /* Performs read operation on the resource */
              if(sc->q_first->req_buf = (u_long*)(data);
.
.
.
}
```

[1] Calls the `lock_read` routine if the `IFF_RUNNING` bit option is set in the `if_flags` field of the `ifp` structure pointer.

The `lock_read` routine takes one argument: a pointer to the complex lock structure `lock`. This lock structure is associated with the resource on which you want to assert a complex lock with read-only access. The `ftaioctl` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer.

Figure 8–1 shows what happens when multiple instances of the `if_fta` kernel module assert a read-only complex lock on the specified code block. As the figure shows, kernel threads from the `if_fta` kernel module that are executing on CPU1, CPU2, and CPU3 assert read-only complex locks on the specified code block. The `lock_read` routine allows multiple kernel threads to have read-only access to the resource at the same time. When a read lock is asserted, the protected resource is guaranteed not to change. In this case, the `cmd_buf` resource is guaranteed not to change.

Figure 8–1: Three Instances of the if_fta Module Asserting a Read-Only Complex Lock



ZK-0961U-AI

8.3.1.2 Asserting a Complex Lock with Write Access

The `lock_write` routine asserts a lock with exclusive write access for the resource that is associated with the specified `lock` structure pointer. After a write lock is asserted, no other kernel thread can gain read or write access to the resource until it is released.

The following code fragment shows a call to `lock_write` by the `if_fta` module's `ftaioctl` routine.

The `ftaioctl` routine is called as the result of an `ioctl` system call.

The `ftaioctl` routine performs the following tasks:

- Determines the type of request
- Executes the request
- Returns data
- Returns the value 0 (zero) to the `ioctl` system call to indicate success

The code fragment also shows the include file that is associated with complex locks, definitions of the `cmd_buf` and `fta_kern_str` structures, the declaration of the complex lock structure in the `fta_kern_str` structure, and the initialization of the complex lock structure by the kernel module's `ftaattach` routine. Section 8.2 provides descriptions of these tasks.

```
#include <kern/lock.h>
.
.

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
};
.
.

struct  fta_kern_str {
.
struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command /*
                            /* request queue */
.
.

};
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.

  /* Tasks to perform controller-specific initialization */
.
.

lock_init(&sc->cmd_buf_q_lock, TRUE);
.
.

 /* Perform other tasks */
}
.
.

ftaioctl(register struct ifnet *ifp,
         unsigned int cmd,
         caddr_t data)
{
    struct fta_kern_str *sc = &fta_kern_str[ifp->if_unit];
.
.

    switch (cmd) {
      case SIOCENABLBACK: {
.
.
```

```
if (ifp->if_flags & IFF_RUNNING) {[1]
    lock_write(&sc->cmd_buf_q_lock);
    sc->q_first->req_buf = (u_long*) (data);
```

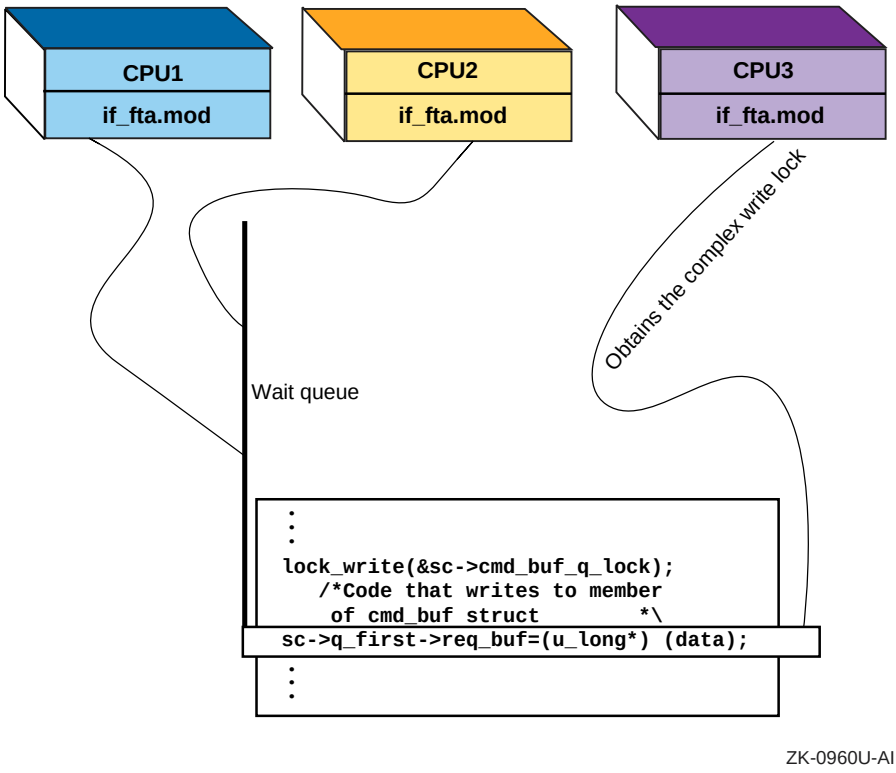
[1] Calls the `lock_write` routine if the `IFF_RUNNING` bit option is set in the `if_flags` field of the `ifp` structure pointer.

The `lock_write` routine takes one argument: a pointer to the complex lock structure `lock`. This lock structure is associated with the resource on which you want to assert a complex lock with write access. The `ftaioctl` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer.

Figure 8–2 shows what happens when multiple instances of the `if_fta` kernel module assert a write complex lock on the specified code block. As the figure shows, kernel threads from the `if_fta` module that are executing on CPU1, CPU2, and CPU3 assert write complex locks on the specified code block. The kernel thread from CPU3 asserts the write complex lock before the kernel threads from CPU1 and CPU2. The kernel thread from CPU3 writes to the `req_buf` field.

The `lock_write` routine blocks (puts to sleep) the kernel threads from CPU1 and CPU2 by placing the requests on a lock queue. This example shows that after `lock_write` successfully asserts a complex write lock, no other kernel thread can gain read or write access to the resource until the resource is released.

Figure 8–2: Three Instances of the if_fta Module Asserting a Write Complex Lock



8.3.2 Releasing a Previously Asserted Complex Lock

After you finish manipulating the resource that is associated with the complex lock, you need to release the lock. To release a complex lock that you previously asserted with a call to `lock_read` or `lock_write`, call the `lock_done` routine. The following code fragment shows a call to `lock_done` by the `if_fta` kernel module's `ftaioctl` routine.

The `ftaioctl` routine is called as the result of an `ioctl` system call.

The `ftaioctl` routine performs the following tasks:

- Determines the type of request
- Executes the request
- Returns data
- Returns the value 0 (zero) to the `ioctl` system call to indicate success

The code fragment also shows the include file that is associated with complex locks, definitions of the `cmd_buf` and `fta_kern_str` structures, the declaration of the complex lock structure in the `fta_kern_str` structure, the initialization of the complex lock structure by the module's `ftaattach` routine, and the assertion of a complex write lock on the code block by the kernel module's `ftaioctl` routine. Section 8.2 and Section 8.3.1.2 provide descriptions of these tasks.

```
.
.
#include <kern/lock.h>
.
.

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
};
.
.

struct  fta_kern_str {
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.

};
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.

  /* Tasks to perform controller-specific initialization */
.
.

lock_init(&sc->cmd_buf_q_lock, TRUE);
.
.

 /* Perform other tasks */
}
.
.

ftaioctl(register struct ifnet *ifp,
         unsigned int cmd,
         caddr_t data)
{
    struct fta_kern_str *sc = &fta_kern_str[ifp->if_unit];
```

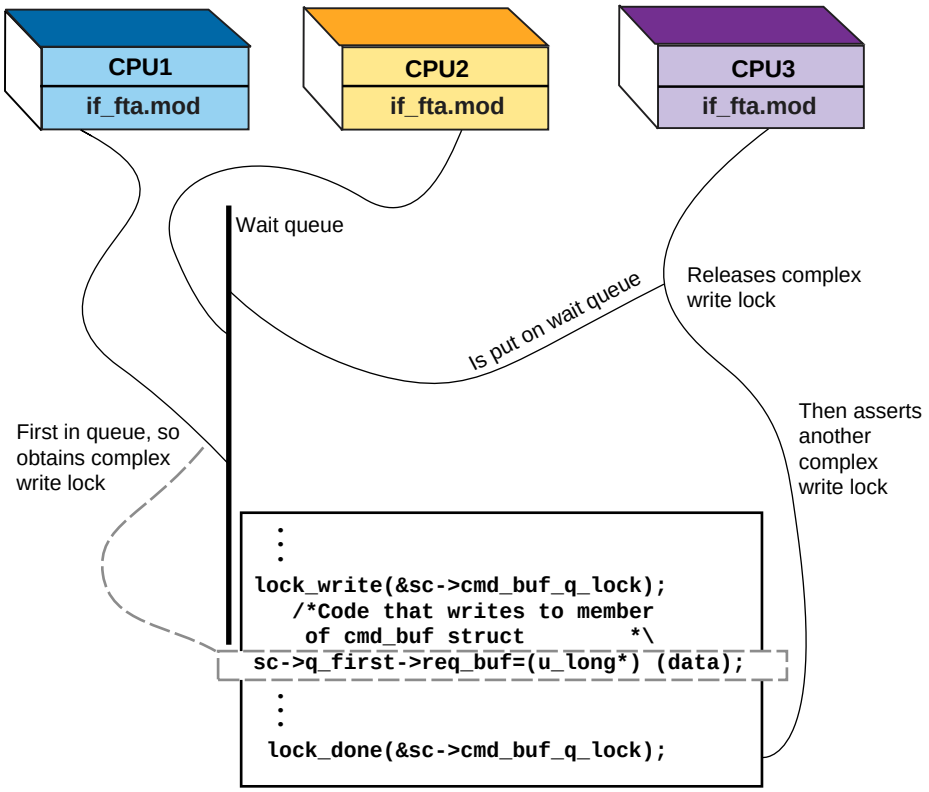
```
.
.
.
    switch (cmd) {
      case SIOCENABLBACK: {
.
.
.
          if (ifp->if_flags & IFF_RUNNING) {
              lock_write(&sc->cmd_buf_q_lock);
              sc->q_first->req_buf = (u_long*) (data);
.
.
.
             lock_done(&sc->cmd_buf_q_lock); 1
.
.
.
}
```

1 Calls the `lock_done` routine to release the complex write lock that `lock_write` previously asserted.

The `lock_done` routine takes one argument: a pointer to the complex lock structure `lock`. This lock structure is associated with the resource on which you want to assert a complex lock with write access. The `ftaioctl` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer.

Figure 8–3 shows what happens when one instance of the `if_fta` module releases a previously asserted complex write lock on the code block that writes to the command buffer queue. As the figure shows, the CPU3 kernel thread releases the complex write lock on the code block that writes to the command buffer queue. The CPU1 and CPU2 kernel threads are blocked, waiting on the wait queue for the complex write lock to be freed. Because the CPU1 kernel thread is first on the wait queue, it now obtains the complex write lock. Furthermore, the figure shows that the CPU3 kernel thread makes another attempt to assert a complex write lock on the code block. This time `lock_write` blocks (puts to sleep) the CPU3 kernel thread by placing it on the wait queue behind the CPU2 kernel thread.

Figure 8–3: One Instance of the if_fta Module Releasing a Complex Write Lock



8.3.3 Trying to Assert a Complex Lock

After you declare and initialize the complex lock data structure, you can try to assert a complex lock with read-only access or a complex lock with write access by calling the `lock_try_read` or `lock_try_write` routine. Unlike the `lock_read` or `lock_write` routines, the `lock_try_read` and `lock_try_write` routines do not block if another kernel thread owns the lock associated with the resource.

The following sections describe how to use these routines.

8.3.3.1 Trying to Assert a Complex Lock with Read-Only Access

To try to assert a complex lock with read-only access, call the `lock_try_read` routine. The `lock_try_read` routine tries to assert a

complex lock (without blocking) with read-only access for the resource that is associated with the specified `lock` structure pointer.

The following code fragment shows a call to `lock_try_read` by the `if_fta` module's `ftaioctl` routine. The code fragment also shows the include file that is associated with complex locks, definitions of the `cmd_buf` and `fta_kern_str` structures, the declaration of the complex lock structure in the `fta_kern_str` structure, and the initialization of the complex lock structure by the module's `ftaattach` routine. Section 8.2 provides descriptions of these tasks. In addition, the code fragment shows a call to `lock_done` if the complex read-only lock is successfully asserted.

```
.
.
#include <kern/lock.h>
.
.

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
};
.
.

struct  fta_kern_str {
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.

};
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.

  /* Tasks to perform controller-specific initialization */
.
.

lock_init(&sc->cmd_buf_q_lock, TRUE);
.
.

 /* Perform other tasks */
}
.
.

ftaioctl(register struct ifnet *ifp,
```

```
        unsigned int cmd,
        caddr_t data)
{
    struct fta_kern_str *sc = &fta_kern_str[ifp->if_unit];
    boolean_t try_ret_val; 1
.
.
.
    switch (cmd) {
      case SIOCENABLBACK: {
.
.
.
         if (ifp->if_flags & IFF_RUNNING) { 2
             try_ret_val = lock_try_read(&sc->cmd_buf_q_lock);
             if (try_ret_val == TRUE) { 3
                if (sc->q_first->req_buf == (u_long*) (data)) {
.
.
.
                lock_done(&sc->cmd_buf_q_lock); 4
                }
             }
          }
.
.
.
             else 5
.
.
.
                /* Code that executes when try_ret_val == FALSE */
.
.
.
     }
.
.
.
          }
.
.
.
}
```

1 Declares a variable to store one of the following return values from the `lock_try_read` routine:

TRUE	The attempt to acquire the read-only complex lock was successful.
FALSE	The attempt to acquire the read-only complex lock was unsuccessful.

2 Calls the `lock_try_read` routine if the `IFF_RUNNING` bit option is set in the `if_flags` field of the `ifp` structure pointer.

The `lock_try_read` routine takes one argument: a pointer to the complex lock structure `lock`. This lock structure is associated with the resource on which you want to try to assert a complex lock with

read-only access. The `ftaioctl` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer.

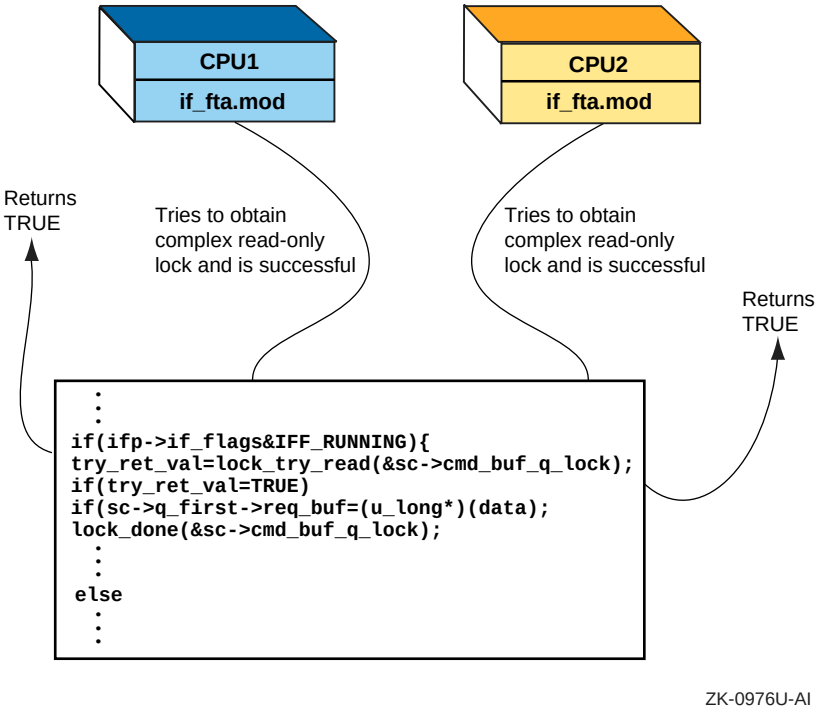
3 If the return from `lock_try_read` is `TRUE`, obtains the read-only complex lock on the code block that performs the read operation.

4 After completing the read operation, releases the read-only complex lock by calling `lock_done`.

5 If the return from `lock_try_read` is `FALSE`, did not obtain the read-only complex lock on the code block that performs the read operation. In this case, it is not necessary to call `lock_done`.

Figure 8–4 shows what happens when two instances of the `if_fta` module attempt to assert a read-only complex lock on the code block that performs a read operation on the resource. As the figure shows, both the CPU1 and CPU2 kernel threads try to assert a read-only complex lock on the code block that performs a read operation on the command buffer queue. Because this is a read-only operation, the CPU1 and CPU2 kernel threads obtain the read-only complex lock, and as a result, `lock_try_read` returns the value `TRUE` in both cases.

Figure 8–4: The if_fta Module Trying to Assert a Complex Read-Only Lock



8.3.3.2 Trying to Assert a Complex Lock with Write Access

To try to assert a complex lock with write access, call the `lock_try_write` routine. The `lock_try_write` routine tries to assert a complex lock (without blocking) with write access for the resource that is associated with the specified `lock` structure pointer.

The following code fragment shows a call to `lock_try_write` by the `if_fta` module's `ftaioctl` routine. The code fragment also shows the include file that is associated with complex locks, definitions of the `cmd_buf` and `fta_kern_str` structures, the declaration of the complex lock structure in the `fta_kern_str` structure, and the initialization of the complex lock structure by the module's `ftaattach` routine. Section 8.2 provides descriptions of these tasks. In addition, the code fragment shows a call to `lock_done` if the complex write lock is successfully asserted.

```
.
.
#include <kern/lock.h>
.
.

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
};
.
.

struct  fta_kern_str {
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.

};
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.

  /* Tasks to perform controller-specific initialization */
.
.

lock_init(&sc->cmd_buf_q_lock, TRUE);
.
.

 /* Perform other tasks */
```

```
}
.
.
.
ftaioctl(register struct ifnet *ifp,
         unsigned int cmd,
         caddr_t data)
{
    struct fta_kern_str *sc = &fta_kern_str[ifp->if_unit];
    boolean_t try_ret_val; 1
.
.
.
    switch (cmd) {
      case SIOCENABLBACK: {
.
.
.
          if (ifp->if_flags & IFF_RUNNING) { 2
              try_ret_val = lock_try_write(&sc->cmd_buf_q_lock);
              if (try_ret_val == TRUE) { 3
                 sc->q_first->req_buf = (u_long*) (data);
.
.
.
                 lock_done(&sc->cmd_buf_q_lock); 4
              }
.
.
.
              else 5
.
.
.
                 /* Code that executes when try_ret_val == FALSE */
.
.
.
     }
.
.
.
          }
.
.
.
}
```

1 Declares a variable to store one of the following return values from the `lock_try_write` routine:

TRUE	The attempt to acquire the write complex lock was successful.
FALSE	The attempt to acquire the write complex lock was unsuccessful.

2 Calls the `lock_try_write` routine if the `IFF_RUNNING` bit option is set in the `if_flags` field of the `ifp` structure pointer.

The lock_try_write routine takes one argument: a pointer to the complex lock structure lock. This lock structure is associated with the resource on which you want to try to assert write access. The ftaioctl routine passes the address of the cmd_buf_q_lock field of the fta_kern_str structure pointer.

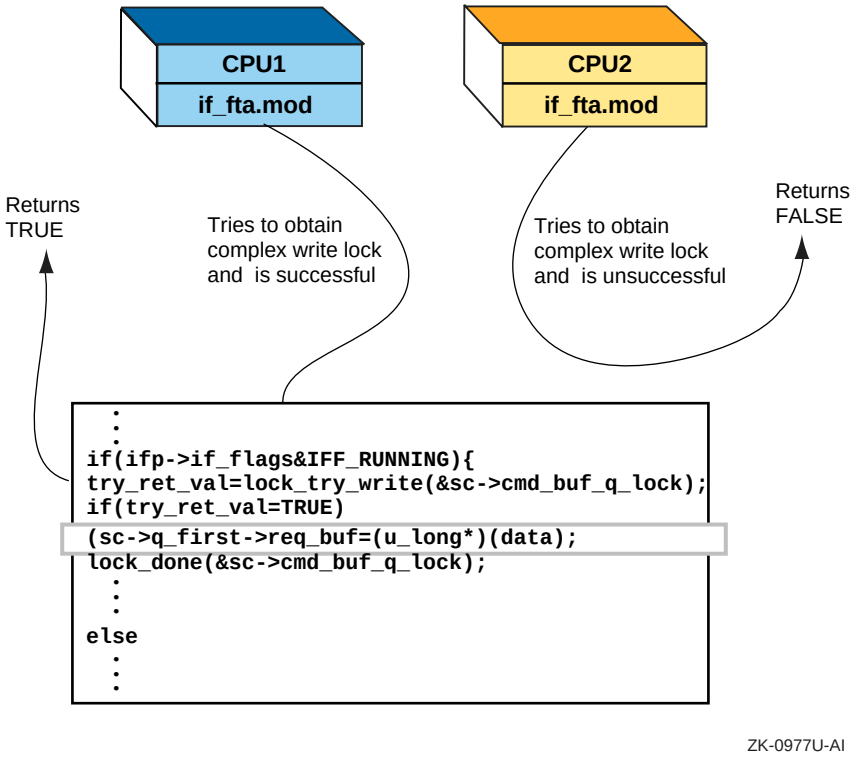
3 If the return from lock_try_write is TRUE, obtains the write complex lock on the code block that performs the write operation.

4 After completing the write operation, releases the write complex lock by calling lock_done.

5 If the return from lock_try_write is FALSE, did not obtain the write complex lock on the code block that performs the write operation. In this case, it is not necessary to call lock_done.

Figure 8–5 shows what happens when two instances of the if_fta module attempt to assert a write complex lock on the code block that performs a write operation on the resource. As the figure shows, both the CPU1 and CPU2 kernel threads try to assert a write complex lock on the code block that performs a write operation on the command buffer queue. The CPU1 kernel thread obtains the write complex lock first and as a result lock_try_write returns the value TRUE. Because the CPU2 kernel thread was not successful in obtaining the write complex lock, lock_try_write immediately returns (does not block the kernel thread) the value FALSE.

Figure 8–5: The if_fta Module Trying to Assert a Complex Write Lock



8.4 Terminating a Complex Lock

After you unlock a complex read or write lock and know that you are finished using the lock for this resource, you can terminate the lock by calling the `lock_terminate` routine. Typically, you terminate any locks in the kernel module's controller (or device) `unattach` routine. These routines are associated with loadable kernel modules. One task that is associated with a controller or device `unattach` routine is to terminate any locks initialized in the kernel module's `attach` routine.

The following code fragment shows a call to `lock_terminate` by the `if_fta` module's `fta_ctlr_unattach` routine. The code fragment also shows the include file that is associated with complex locks, definitions of the `cmd_buf` and `fta_kern_str` structures, the declaration of the complex lock structure in the `fta_kern_str` structure, and the initialization of the complex lock structure by the kernel module's `ftaattach` routine. Section 8.2 provides descriptions of these tasks. In addition, the code fragment shows calls to `lock_write` and `lock_done`.

```
.
.
#include <kern/lock.h>
.
.

struct cmd_buf {
        u_long *req_buf;
        u_long *rsp_buf;
        short  timeout;
        struct cmd_buf *next;
};
.
.

struct  fta_kern_str {
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the command */
                            /* request queue */
.
.

};
.
.

ftaattach(struct controller *ctlr)
{
        struct fta_kern_str *sc = &fta_kern_str[ctlr->ctlr_num];
.
.

  /* Tasks to perform controller-specific initialization */
.
.

lock_init(&sc->cmd_buf_q_lock, TRUE);
.
.

 /* Perform other tasks */
}
.
.

ftaioctl(register struct ifnet *ifp,
         unsigned int cmd,
         caddr_t data)
{
    struct fta_kern_str *sc = &fta_kern_str[ifp->if_unit];
.
.

    switch (cmd) {
      case SIOCENABLBACK: {
.
.

          if (ifp->if_flags & IFF_RUNNING) {
              lock_write(&sc->cmd_buf_q_lock);
              sc->q_first->req_buf = (u_long*) (data);
```

```
.
.
.
            lock_done(&sc->cmd_buf_q_lock);
.
.
.
}
.
.
.
fta_ctlr_unattach(struct bus *bus,
                  struct controller *ctlr)
{
        register int unit = ctlr->ctlr_num;

        if ((unit > num_fta) || (unit < 0) {
                return(1);
        }

        if (fta_is_dynamic == 0) {
               return(1);
}
/* Performs controller unattach tasks */
.
.
.
                lock_terminate(&sc->cmd_buf_q_lock); 1
.
.
.
}
```

1 Calls the `lock_terminate` routine to determine that the `if_fta` module is permanently done using this complex lock.

The `lock_terminate` routine takes one argument: a pointer to the complex lock structure `lock`. In this call, the `fta_ctlr_unattach` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer. In calling `lock_terminate`, the `if_fta` module must not reference this complex lock again.

9

Kernel Threads

This chapter discusses the following topics associated with kernel threads:

- The advantages of using kernel threads (Section 9.1)
- Kernel threads execution (Section 9.1.1)
- Issues related to using kernel threads (Section 9.1.2)
- Kernel threads operations (Section 9.1.3)
- The `thread` and `task` data structures (Section 9.2)

In addition, this chapter discusses the routines that allow you to perform kernel thread operations. Specifically, these routines allow you to:

- Create and start a kernel thread (Section 9.3)
- Block (put to sleep) a kernel thread (Section 9.4)
- Unblock (wake up) kernel threads (Section 9.5)
- Terminate a kernel thread (Section 9.6)
- Set a timer for the current kernel thread (Section 9.7)

9.1 Using Kernel Threads in Kernel Modules

A thread is a single, sequential flow of control within a program. Within a single thread is a single point of execution. Applications use threads to improve their performance (throughput, computational speed, and responsiveness). To start, terminate, delete, and perform other operations on threads, the application programmer calls the routines that the POSIX Threads Library provides.

The term *kernel thread* distinguishes between the threads that applications use. A kernel thread is a single sequential flow of control within a kernel module or other systems-based program. The kernel module or other systems-based program uses the routines (instead of a threads library package such as the POSIX Threads Library) to start, terminate, delete, and perform other kernel thread operations.

Kernel threads execute within (and share) a single address space. Therefore, kernel threads read from and write to the same memory locations.

You use kernel threads to improve the performance of a kernel module. Multiple kernel threads are useful in a multiprocessor environment, where kernel threads run concurrently on separate CPUs. However, multiple kernel threads also improve kernel module performance on single-processor systems by permitting the overlap of input, output, or other slow operations with computational operations.

Kernel threads allow kernel modules to perform other useful work while waiting for a device to produce its next event, such as the completion of a disk transfer or the receipt of a packet from the network.

Typically, you use kernel threads in kernel modules when:

- The kernel module must perform a long operation

  One example of a long operation is the reset sequences for a multistep device.

  One reason for creating a kernel thread to perform a long operation is to prevent the kernel module from running at a high interrupt priority level (IPL) for long periods of time.

- The resource or resources for that operation are not available

  This situation refers to allocating memory or accessing address space that might cause a page fault.

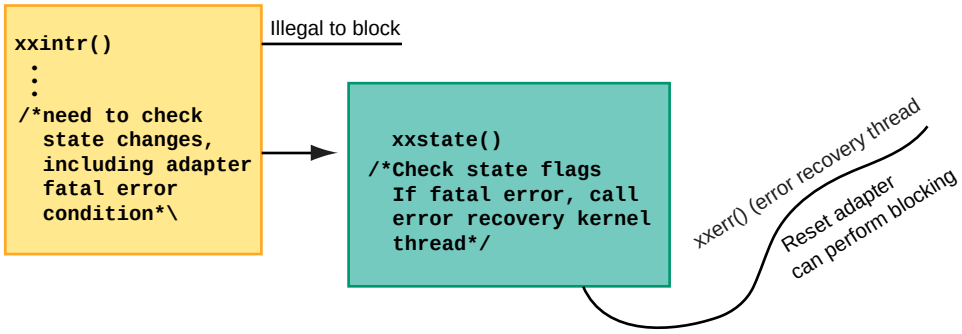
- The operation to be performed on the resource (for example, blocking) is illegal.

  For example, access to a data item is not allowed at an elevated IPL, such as the `proc` structure.

Figure 9–1 shows one example of the previously described situations. As the figure shows, a kernel module must verify a number of device state changes. One of these device state changes searches for an adapter fatal error condition. If the fatal error condition occurs, the kernel module must reset the adapter. The code that resets the adapter must block to accomplish the adapter reset operation. Furthermore, the only time this error can occur is during a device interrupt.

It is not legal to block in an interrupt service routine. Therefore, the figure shows that the interrupt service routine for the kernel module calls an `xxstate` routine that handles all of the state changes. This routine creates a kernel thread called `xxerr` that starts up when the adapter becomes operational. The job of this kernel thread is to reset the adapter when a fatal error occurs. It is legal for this kernel thread to perform blocking operations.

Figure 9–1: Using Kernel Threads in a Kernel Module



ZK-1015U-AI

9.1.1 Kernel Threads Execution

You can view multiple kernel threads in a program as executing simultaneously. However, you cannot make any assumptions about the relative start or finish times of kernel threads or the sequence in which they execute. You can influence the scheduling of kernel threads by setting scheduling and policy priority.

Each kernel thread has its own unique thread identifier. This thread identifier is a pointer to the `thread` data structure for the kernel thread. The kernel threads creation routines return this `thread` data structure pointer to the kernel module after they successfully create and start the kernel thread. Kernel modules use this pointer as a handle to a specific kernel thread in calls to other kernel thread routines.

A kernel thread changes states during the course of its execution and is always in one of the following states:

- Waiting

  The kernel thread is not eligible to execute because it is synchronizing with another kernel thread or with an external event, such as I/O.

- Ready

  The kernel thread is eligible for a CPU to execute.

- Running

  A CPU is currently executing a kernel thread.

- Terminated

  The kernel thread has completed all of its work.

9.1.2 Issues Related to Using Kernel Threads

When you design and code a kernel module that uses the kernel thread routines, consider the following issues:

- Interplay among kernel threads

  Using kernel threads can simplify the coding and designing of a kernel module. However, you need to be sure that the synchronization and interplay among kernel threads are correct. You use simple and complex locks to synchronize access to data.

- Race conditions

  A race condition is a programming error that causes unpredictable and erroneous program behavior. Specifically, the error occurs when two or more kernel threads perform an operation and the result of the operation depends on unpredictable timing factors, for example, when each kernel thread executes and waits and when each kernel thread completes the operation.

- Deadlocks

  A deadlock is a programming error that causes two or more kernel threads to be blocked indefinitely. Specifically, the error occurs when a kernel thread holds a resource while it waits for a resource that another kernel thread holds and that kernel thread is also waiting for the first kernel thread's resource.

- Priority inversion

  Priority inversion occurs when the interaction among three or more kernel threads blocks the highest-priority kernel thread from executing. For example, a high-priority kernel thread waits for a resource that a low-priority kernel thread has locked, and the low-priority kernel thread waits while a middle-priority kernel thread executes. The high-priority kernel thread is made to wait while a kernel thread of lower priority (the middle-priority kernel thread) executes.

  To avoid priority inversion, associate a priority with each resource that is at least as high as the highest-priority kernel thread that will use it, and force any kernel thread that uses that object to first raise its priority to that of the object.

9.1.3 Kernel Threads Operations

Table 9–1 lists the routines for kernel threads, and describes the operations they perform.

Table 9–1: Summary of Operations That Kernel Thread Routines Perform

Routines	Description
Creating kernel threads	
`kernel_isrthread`	Starts a fixed-priority kernel thread dedicated to interrupt service.
`kernel_isrthread_w_arg`	Starts a fixed-priority kernel thread with an argument.
`rad_kernel_isrthread`	Starts a fixed-priority thread on a specified **RAD**; takes an argument for the thread that can be passed as NULL.
`kernel_thread`	Starts a timeshare kernel thread without a calling argument.
`kernel_thread_w_arg`	Starts a timeshare kernel thread with a calling argument passed in.
`rad_kernel_thread`	Starts a timeshare kernel thread on a specified RAD; takes an argument that can be passed as NULL.
Blocking kernel threads	
`assert_wait_mesg`	Asserts that the current kernel thread is about to block (sleep).
`thread_block`	Blocks (puts to sleep) the current kernel thread.
Unblocking kernel threads	
`thread_wakeup`	Wakes up all kernel threads waiting for the specified event.
`thread_wakeup_one`	Wakes up the first kernel thread waiting on a channel.
Terminating kernel threads	
`thread_terminate`	Prepares to stop or stops execution of the specified kernel thread.
`thread_halt_self`	Handles asynchronous traps for self-terminating kernel threads.
Miscellaneous	
`current_task`	Returns a pointer to the `task` structure for the currently running kernel thread.
`thread_set_timeout`	Sets a timer for the current kernel thread.

9.2 Using the thread and task Data Structures

This section discusses the two data structures that the kernel thread routines use: `thread` and `task`. The `thread` data structure contains kernel thread information. Kernel modules typically use the `wait_result` field (along with the `current_thread` routine) to determine the result of the wait. The `/usr/sys/include/kern/thread.h` header file shows a `typedef` statement that assigns the alternate name `thread_t` for a pointer to the `thread` structure. Many of the kernel thread routines operate on these pointers to `thread` structures.

The `thread` structure is an opaque data structure; that is, the operating system, not the user of kernel threads, references and manipulates all associated fields (except for the `wait_result` field) .

The `task` data structure contains task-related information. The `/usr/sys/include/kern/task.h` header file shows a `typedef` statement that assigns the alternate name `task_t` for a pointer to the `task` structure. Some of the kernel thread routines require that you pass a pointer to the `task` structure.

The `task` structure is an opaque data structure; that is, the operating system, not the user of kernel threads, references and manipulates all of its associated fields.

9.3 Creating and Starting a Kernel Thread

You can create and start a kernel thread with any of the functions listed in the *Creating kernel threads* section of Table 9–1.

Section 9.3.1 and Section 9.3.2 show examples using `kernel_thread_w_arg` and `kernel_isrthread`. You can also adapt these examples to create a kernel timeshare thread without passing arguments and/or on a specific RAD. Additionally, you can create a `kernel_isrthread` with an argument and/or on a specific RAD.

Note

All of the kernel thread functions listed in Table 9–1 operate as wrappers around the corresponding `rad_kernel_thread` or `rad_kernel_isthread` function. Kernel threads that were created with the older kernel thread functions are created on the current RAD for Non-Uniform Memory Access (NUMA) systems or on the only RAD for non-NUMA systems. Both the `rad_kernel_thread` function and the `rad_kernel_isthread` function take an argument to the thread function that can be

specified as NULL if your thread function does not need an argument.

9.3.1 Creating and Starting a Kernel Thread with an Argument and Timeshare Scheduling Policy

To create and start a kernel thread with an argument and timeshare policy, call the `kernel_thread_w_arg` routine. If you do not need an argument for the thread, call the `kernel_thread` routine. If you want to create the thread on a specific RAD in a NUMA system, call the `rad_kernel_thread` routine.

The `kernel_thread_w_arg` routine creates and starts a kernel thread in a specified task at a specified **entry point**. It passes a specified argument to the newly created kernel thread and creates and starts a kernel thread with timeshare scheduling. A kernel thread that is created with timeshare scheduling means that its priority degrades if it consumes an inordinate amount of CPU resources.

Note

Ensure that a kernel module calls `kernel_thread_w_arg` only for long-running tasks.

The following code fragment shows a call to `kernel_thread_w_arg` by the `if_fta` module's `fta_transition_state` routine. The `fta_transition_state` routine changes the state of the kernel module by performing certain fixed functions for any given state.

```
#include <kern/thread.h>[1]
.
.
.
#define ADAP "fta"
.
.
.
extern task_t first_task;[2]
.
.
.
struct  fta_kern_str {
.
.
.
short reinit_thread_started; /* reinit thread running? */
.
.
.
};[3]
.
.
.
struct  ifnet {
```

```
.
.
.
short if_unit; /* subunit for lower-level driver */
.
.
.
};
.
.
.
fta_transition_state(struct fta_kern_str *sc,
                     short unit,
                     short state)
{
.
.
.
    switch(state) {
.
.
.
      case PI_OPERATIONAL: {
          int s;
          NODATA_CMD *req_buff;
          thread_t thread; [4]

          if (sc->reinit_thread_started == FALSE) { [5]

              thread = kernel_thread_w_arg(first_task,
                                           fta_error_recovery,
                                           (void *)sc); [6]
              if (thread == NULL) { [7]
                  printf("%s%d: Cannot start error recovery thread.\n",
                         ADAP, ifp->if_unit);
              }
              sc->reinit_thread_started = TRUE;
          }
}
```

[1] Includes the `/usr/sys/include/kern/thread.h` header file. The `thread.h` file defines structures that the kernel thread routines use.

[2] Declares a pointer to a `task` structure and calls it `first_task`. Every kernel thread must be part of a task. You pass this pointer to the `kernel_thread_w_arg` routine.

[3] Defines an `fta_kern_str` data structure. The example shows only the field that is related to the discussion of the `kernel_thread_w_arg` routine.

[4] Declares a pointer to a `thread` structure and calls it `thread`. This variable stores the `thread` structure pointer that `kernel_thread_w_arg` returns.

[5] If the reinitialized kernel thread evaluates to `FALSE` (the reinit kernel thread is not running), it calls the `kernel_thread_w_arg` routine.

[6] Calls the `kernel_thread_w_arg` routine.

The `kernel_thread_w_arg` routine takes three arguments:

- The first argument specifies a pointer to a `task` structure. The `task` structure is maintained only for source compatibility only and should be ignored. Instead, specify NULL.

  In the example, the `fta_transition_state` routine passes a `task` structure called `first_task`.

- The second argument specifies a pointer to a routine that is the entry point for the newly created kernel thread. In this call, the entry point for the newly created kernel thread is the `fta_error_recovery` routine. The `fta_error_recovery` routine is a kernel thread that starts up when the adapter becomes operational. This kernel thread is responsible for resetting the adapter in the event of a fatal error.

- The third argument specifies an argument that `kernel_thread_w_arg` passes to the entry point that the second argument specifies. In this call, the `fta_transition_state` routine passes a pointer to the `fta_kern_str` structure. The `fta_error_recovery` routine performs a variety of tasks that require the `fta_kern_str` structure.

7 Upon successful completion, `kernel_thread_w_arg` returns a pointer to the `thread` structure for the kernel thread that started at the specified entry point. Kernel modules can use this pointer as a handle to a specific kernel thread in calls to other kernel thread routines.

The `fta_transition_state` routine inspects the return. If the return is `NULL`, `kernel_thread_w_arg` did not create the error recovery kernel thread. The `fta_transition_state` routine calls `printf` to display an appropriate message on the console terminal. If the return is not `NULL`, `fta_transition_state` sets the `reinit_thread_started` field to the value `TRUE` to indicate that the error recovery kernel thread is started.

9.3.2 Creating and Starting a Fixed-Priority Kernel Thread Dedicated to Interrupt Service

To create and start a fixed-priority kernel thread that is dedicated to interrupt service, call the `kernel_isrthread` routine. If you need an argument for the thread, call the `kernel_isrthread_w_arg` routine. If you want to create the thread on a specific RAD in a NUMA system, call the `rad_kernel_isrthread` routine.

The following example code fragment shows a call to `kernel_isrthread` by the `if_fta` module's `ftaprobe` routine. This kernel thread handles only interrupt service requests in the specified task and at the specified priority level. Make sure that a kernel module always attaches a kernel thread to the first task.

In the example code fragment, the `ftaprobe` routine determines if the adapter exists, fills in a variety of register values, and initializes a variety of descriptors.

```
.
.
.
#include <kern/thread.h> 1
.
.
.

extern task_t first_task; 2
.
.
.

ftaprobe(io_handle_t reg,
         struct  controller *ctlr)
{
.
.
.


           thread = kernel_isrthread(first_task,
                                     fta_rec_intr,
                                     BASEPRI_SYSTEM); 3
.
.
.

}
```

1 Includes the `/usr/sys/include/kern/thread.h` header file. The `thread.h` file defines structures that kernel thread routines use.

2 Declares a pointer to a `task` structure and calls it `first_task`. Every kernel thread must be part of a task. You pass this pointer to the `kernel_isrthread` routine.

3 Calls the `kernel_isrthread` routine.

The `kernel_isrthread` routine takes three arguments:

- The first argument specifies a pointer to a `task` structure. The `task` structure is maintained for source compatibility only and should be ignored. Instead, specify NULL.

  In the example, the `ftaprobe` routine passes a `task` structure called `first_task`.

- The second argument specifies a pointer to a routine that is the entry point for the newly created kernel thread. In this call, the entry point for the newly created kernel thread is the `fta_rec_intr` routine. The `fta_rec_intr` routine is a kernel thread that starts up when the kernel module discovers a receive type device interrupt. This kernel thread is responsible for handling the receive type interrupt.

- The third argument specifies the scheduling priority level for the newly created kernel thread.

  The following priority usage table describes the possible scheduling priorities. The first column shows a range of priorities. The second

column shows an associated scheduling priority constant that is defined in <src/kernel/kern/sched.h> (if applicable). The third column describes the usage of the priority ranges. To specify a scheduling priority of 38, you pass the constant BASEPRI_SYSTEM, as shown in the example. To specify a scheduling priority of 33, you can pass the following: BASEPRI_HIGHEST + 1.

Priority	Constant	Usage
0—31	N/A	Real-time kernel threads
32—38	BASEPRI_HIGHEST — BASEPRI_SYSTEM	Operating system kernel threads
44—64	BASEPRI_USER — BASEPRI_LOWEST	User kernel threads

9.4 Blocking (Putting to Sleep) a Kernel Thread

The routines that you use to block (put to sleep) a kernel thread depend on whether or not the block (sleep) can be interrupted. For interruptible sleep (that is, the kernel thread can take asynchronous signals), you must call the symmetric multiprocessor (SMP) sleep call, mpsleep (see Section 9.4.2).

For uninterruptible sleep, use one of the following routines:

- assert_wait_mesg

  Call this routine to assert that the current kernel thread is about to block until some specified event occurs. You use this routine with the thread_block routine, which actually blocks (puts to sleep) the current kernel thread.

- thread_block

  Call this routine to block the current kernel thread and select the next kernel thread to start.

These routines are described in the following sections.

9.4.1 Asserting That the Current Kernel Thread Is About to Block Until the Specified Event Occurs

To assert that the current kernel thread is about to block until some specified event occurs, call the assert_wait_mesg routine. To actually block (put to sleep) the current kernel thread, call thread_block.

The following code fragment shows a call to assert_wait_mesg and thread_block by the if_fta module's fta_error_recovery routine. The fta_error_recovery routine is a kernel thread that starts up when the adapter becomes operational. This kernel thread resets the adapter if a

fatal error occurs. The code fragment also shows the code that contains the call to `kernel_thread_w_arg`, which calls `fta_error_recovery`.

```
.
.
.
#include <kern/thread.h> 1
.
.
.

#define ADAP "fta"
.
.
.

extern task_t first_task; 2
.
.
.

struct  fta_kern_str {
.
.
.

short reinit_thread_started; /* reinit thread running? */
.
.
.

short error_recovery_flag; /* flag to wake up a process */
.
.
.

}; 3
.
.
.

struct  ifnet {
.
.
.

short if_unit; /* subunit for lower-level driver */
.
.
.

};
.
.
.

fta_transition_state(struct fta_kern_str *sc,
                     short unit,
                     short state)
{
.
.
.

    switch(state) {
.
.
.

      case PI_OPERATIONAL: {
          int s;
          NODATA_CMD *req_buff;
          thread_t thread; 4

          if (sc->reinit_thread_started == FALSE) { 5

              thread = kernel_thread_w_arg(first_task,
                                           fta_error_recovery,
                                           (void *)sc); 6
```

```
            if (thread == NULL) { 7
                printf("%s%d: Cannot start error recovery thread.\n",
                       ADAP, ifp->if_unit);
            }
            sc->reinit_thread_started = TRUE;
        }
.
.
.

void fta_error_recovery(struct fta_kern_str *sc) 8
{
    struct ifnet *ifp;


    /*
     * Collect the argument left by the kernel_thread_w_arg().
     */
    ifp = &sc->is_if;

    for(;;) { 9
        assert_wait_mesg((vm_offset_t)&sc->error_recovery_flag,
                         TRUE,"ftaerr"); 10
        thread_block(); 11

/* Performs tasks to reset the adapter */
.
.
.

}
.
.
.

}
```

1 Includes the `/usr/sys/include/kern/thread.h` header file. The `thread.h` file defines structures that kernel thread routines use.

2 Declares a pointer to a `task` structure and calls it `first_task`. Every kernel thread must be part of a task. You pass this pointer to the `kernel_thread_w_arg` routine.

3 Defines an `fta_kern_str` data structure. The example shows only the fields that are related to the discussion of the `kernel_thread_w_arg`, `assert_wait_mesg`, and `thread_block` routines.

4 Declares a pointer to a `thread` structure and calls it `thread`. This variable stores the `thread` structure pointer that `kernel_thread_w_arg` returns.

5 If the reinitialized kernel thread evaluates to `FALSE` (the `reinit` kernel thread is not running), calls the `kernel_thread_w_arg` routine.

6 Calls the `kernel_thread_w_arg` routine.

The `kernel_thread_w_arg` routine takes three arguments:

- The first argument specifies a pointer to a `task` structure. This pointer identifies the task in which the `kernel_thread_w_arg` routine starts the newly created kernel thread. In this call, the

fta_transition_state routine passes a task structure called first_task.

- The second argument specifies a pointer to a routine that is the entry point for the newly created kernel thread. In this call, the entry point for the newly created kernel thread is the fta_error_recovery routine. The fta_error_recovery routine is a kernel thread that starts up when the adapter becomes operational. This kernel thread is responsible for resetting the adapter in the event of a fatal error.
- The third argument specifies an argument that kernel_thread_w_arg passes to the entry point that the second argument specifies. In this call, the fta_transition_state routine passes a pointer to the fta_kern_str structure. The fta_error_recovery routine performs a variety of tasks that require the fta_kern_str structure.

7 Upon successful completion, kernel_thread_w_arg returns a pointer to the thread structure for the kernel thread started at the specified entry point. Kernel modules can use this pointer as a handle to a specific kernel thread in calls to other kernel thread routines.

The fta_transition_state routine checks the return. If the return is NULL, kernel_thread_w_arg did not create the error recovery kernel thread. The fta_transition_state routine calls printf to display an appropriate message on the console terminal. If the return is not NULL, fta_transition_state sets the reinit_thread_started field to the value TRUE to indicate that the error recovery kernel thread is started.

8 The fta_error_recovery routine is a kernel thread that starts up when the adapter becomes operational. This kernel thread resets the adapter if a fatal error occurs.

A fatal error requires resetting the adapter; this error is discovered during a device interrrupt. It is necessary to block in the interrupt service routine while resetting the adapter. Because it is not legal to block in an interrupt service routine, the fta_transition_state calls this kernel thread to perform the reset operation on the adapter.

The kernel_thread_w_arg routine passes the kern_str structure pointer to fta_error_recovery.

9 Sets up an infinite loop that executes when the adapter becomes operational.

10 Calls the assert_wait_mesg routine to assert that the current kernel thread is about to block (sleep).

The assert_wait_mesg routine takes three arguments:

- The first argument specifies the event to associate with the current kernel thread. In this call, the event that is associated with the current kernel thread is stored in the `error_recovery_flag` field.
- The second argument specifies a Boolean value that indicates how the kernel thread is awakened. You can pass one of the following values:

TRUE	The current kernel thread is interruptible. This value means that a signal can awaken the current kernel thread.
FALSE	The current kernel thread is not interruptible. This value means that only the specified event can awaken the current kernel thread.

  In this call, the value TRUE is passed.
- The third argument specifies a mnemonic for the type of wait. The `/bin/ps` command uses this mnemonic to print out more meaningful messages about a process. In this call, the `fta_error_recovery` routine passes the string `ftaerr`.

The `assert_wait_mesg` routine does not return a value.

11 Calls the `thread_block` routine. The `thread_block` routine blocks (puts to sleep) the current kernel thread and selects the next kernel thread to start (run). The routine schedules the next kernel thread onto this CPU.

The `thread_block` routine does not return a value.

9.4.2 Using the Symmetric Multiprocessor Sleep Routine

To block the current kernel thread, call the `mpsleep` routine — the symmetric multiprocessor (SMP) sleep call. The following code fragment shows a call to `mpsleep` by the `if_fta` module's `fta_error_recovery` routine. The `fta_error_recovery` routine is a kernel thread that starts when the adapter becomes operational. This kernel thread resets the adapter if a fatal error occurs. The code fragment also shows the use of a simple lock with the `mpsleep` routine.

```
.
.
.
struct  fta_kern_str {
.
.
.
short error_recovery_flag; /* flag to wake up a process */
.
.
.
```

```
int is_state; 1
simple_lock_data_t lk_fta_kern_str; 2
.
.
.
};
.
.
.
void fta_error_recovery(struct fta_kern_str *sc)
{
    struct ifnet *ifp;


    /*
     * Collect the argument left by the kernel_thread_w_arg().
     */
    ifp = &sc->is_if;
    simple_lock (&sc->lk_fta_kern_str); 3
    while (sc->is_state == RUN_NOT) { 4

    for(;;) { 5
        mpsleep ((vm_offset_t)&sc->error_recovery_flag, PCATCH,
                 "ftaerr", 0, &sc->lk_fta_kern_str,
                 MS_LOCK_SIMPLE | MS_LOCK_ON_ERROR)) 6

/* Performs tasks to reset the adapter */
.
.
.
            }
    }
}
.
.
.
```

1 Declares a field to hold state options.

2 Declares a simple lock structure pointer as a field of the `fta_kern_str` structure to protect the integrity of the data that is stored in the fields of this structure. Assume that this simple lock was initialized in the example kernel module's `attach` routine. The `fta_error_recovery` routine passes this simple lock structure pointer to the `mpsleep` routine.

3 Calls the `simple_lock` routine to assert an exclusive access on the following code block.

4 While the `is_state` option is equal to the `RUN_NOT` option, execute the for loop.

5 Sets up an infinite loop that executes when the `is_state` option is equal to the `RUN_NOT` option.

6 Calls the `mpsleep` routine to block (put to sleep) the current kernel thread.

The `mpsleep` routine takes six arguments:

- A `channel` argument

The channel argument specifies an address to associate with the calling kernel thread to be put to sleep. In this call, the address (or event) that is associated with the current kernel thread is stored in the error_recovery_flag field.

- A pri argument

  The pri argument specifies whether the sleep request is interruptible. Setting this argument to the PCATCH option causes the process to sleep in an interruptible state (that is, the kernel thread can take asynchronous signals). Not setting the PCATCH option causes the process to sleep in an uninterruptible state (that is, the kernel thread cannot take asynchronous signals).

  In this call, fta_error_recovery passes the value PCATCH.

- A wmesg argument

  The wmesg argument specifies the wait message.

  In this call, fta_error_recovery passes the string ftaerr.

- A timo argument

  The timo argument specifies the maximum amount of time for the kernel thread to block (sleep). If you pass the value 0 (zero), mpsleep assumes there is no timeout.

  In this call, fta_error_recovery passes the value 0 (zero) to indicate there is no timeout.

- A lockp argument

  The lockp argument specifies a pointer to a simple or complex lock structure. You pass a simple or complex lock structure pointer if you want to release the lock. If you do not want to release a lock, pass the value 0 (zero).

  In this call, fta_error_recovery passes the address of the simple lock.

- A flags argument

  The flags argument specifies the lock type. You can pass the bitwise inclusive OR of the valid lock bits that are defined in /usr/sys/include/sys/param.h.

  In this call, fta_error_recovery passes the bitwise inclusive OR of the lock bits MS_LOCK_SIMPLE (calls mpsleep with a simple lock asserted) and MS_LOCK_ON_ERROR (forces mpsleep to relock the lock on failure). You specify these bits only if you pass a simple or complex lock.

The mpsleep routine blocks (puts to sleep) the current kernel thread until a wakeup is issued on the address that you specified in the

channel argument. The kernel thread blocks a maximum of timo divided by hz seconds. The value 0 (zero) means there is no timeout.

If you pass the PCATCH option to the pri argument, mpsleep examines signals before and after blocking. Otherwise, mpsleep does not examine signals.

The mpsleep routine allows you to specify a pointer to a simple or complex lock structure for some resource. This routine unlocks this resource prior to blocking. The flags argument specifies the lock type. The mpsleep routine releases the lock when the current kernel thread successfully performs an assert wait on the specified channel.

The mpsleep routine returns the value 0 (zero) if awakened (success) and returns EWOULDBLOCK if the timeout specified in the timo argument expires (failure). On success, mpsleep relocks the lock if you did not set MS_LOCK_NO_RELOCK in flags. On failure, it leaves the lock unlocked. If you set the flags argument to MS_LOCK_ON_ERROR, mpsleep relocks the lock on failures.

9.5 Unblocking (Awakening) Kernel Threads

You can unblock (awaken) a kernel thread with the following routines:

- thread_wakeup_one

  Call this routine to unblock the first kernel thread on the specified event.

- thread_wakeup

  Call this routine to unblock all kernel threads on the specified event.

Note

If the thread was blocked using the mpsleep routine (see Section 9.4.2), use the wakeup routine passing the same "channel" argument that is passed to mpsleep. This approach is the same as the technique for blocking and awakening a process (see Section 5.3.2 and Section 5.3.3). Blocking and awakening a process is actually blocking and awakening a thread, perhaps the only thread, of the process.

The following code fragment compares the calls to thread_wakeup_one and thread_wakeup by the if_fta module's ftaintr routine:

```
ftaintr(int unit)
{
.
.
.
        fta_transition_state(sc, unit, PI_OPERATIONAL); 1
```

```
.
.
.
/*****************************************************
 * Code fragment 1: Shows call to thread_wakeup_one    *
 *****************************************************/
.
.
.
   thread_wakeup_one((vm_offset_t)&sc->error_recovery_flag); 2
}
```

1 This code fragment shows the call to `fta_transition_state`. The `fta_transition_state` routine changes the state of the kernel module by performing certain fixed functions for any given state.

After `fta_transition_state` performs its tasks, it returns to `ftaintr`, which calls `thread_wakeup_one`. This routine takes an `event` as the first argument.

2 The code fragment shows that the first argument for each of the routines specifies the event to associate with the current kernel thread. It passes the address of the value that is stored in the `error_recovery_flag` field.

The kernel module's `fta_error_recovery` routine is the kernel thread that was created and started to perform error recovery tasks. The `fta_error_recovery` routine blocked on the event stored in the `error_recovery_flag` field.

```
ftaintr(int unit)
{
.
.
.
        fta_transition_state(sc, unit, PI_OPERATIONAL); 1
.
.
.
/*****************************************************
 * Code fragment 2: Shows call to thread_wakeup        *
 *****************************************************/
.
.
.
   thread_wakeup((vm_offset_t)&sc->error_recovery_flag); 2
}
```

1 This code fragments shows the call to `fta_transition_state`. The `fta_transition_state` routine changes the state of the kernel module by performing certain fixed functions for any given state.

After `fta_transition_state` performs its tasks, it returns to `ftaintr`, which calls `thread_wakeup`. This routine takes an `event` as the first argument.

2 The code fragment shows that the first argument for each of the routines specifies the event to associate with the current kernel thread. It passes

the address of the value that is stored in the error_recovery_flag field.

The kernel module's fta_error_recovery routine is the kernel thread that was created and started to perform error recovery tasks. The fta_error_recovery routine blocked on the event stored in the error_recovery_flag field.

The thread_wakeup_one routine wakes up only the first kernel thread in the hash chain waiting for the event that the event argument specifies. This routine is actually a convenience wrapper for the thread_wakeup_prim routine with the one_thread argument set to TRUE (wake up only the first kernel thread) and the result argument set to THREAD_AWAKENED (wakeup is normal).

The thread_wakeup routine wakes up all kernel threads that are waiting for the event that the event argument specifies. This routine is actually a convenience wrapper for the thread_wakeup_prim routine with the one_thread argument set to FALSE (wake up all kernel threads) and the result argument set to THREAD_AWAKENED (wakeup is normal).

9.6 Terminating a Kernel Thread

To terminate a kernel thread, call the thread_terminate routine. The thread_terminate routine prepares to stop or permanently stops execution of the specified kernel thread. You created and started this kernel thread in a previous call to the kernel_isrthread or kernel_thread_w_arg routine. These routines return a pointer to the thread structure that is associated with the newly created and started kernel thread. Kernel modules use this pointer as a handle to identify the specific kernel thread that thread_terminate stops executing.

Typically, a kernel thread terminates itself. However, one kernel thread can terminate another kernel thread. A kernel thread that terminates itself must call thread_halt_self immediately after the call to thread_terminate, because thread_terminate only prepares the self-terminating kernel thread to stop execution. The thread_halt_self routine completes the work that is needed to stop execution by performing the appropriate cleanup work of the self-terminating kernel thread.

You do not need to terminate every kernel thread that you create. Do not terminate a kernel thread that is waiting for some event. The basic rule is — terminate only those kernel threads that you do not need anymore. For example, if a dynamically configured kernel module uses kernel threads, terminate them in the CFG_OP_UNCONFIGURE entry point of the loadable kernel module's configure routine. The kernel threads are no longer needed after the kernel module is unconfigured.

The `thread_terminate` routine (for kernel threads that terminate other kernel threads) not only permanently stops execution of the specified kernel thread, but it also frees any resources that are associated with the kernel thread; therefore, this kernel thread can no longer be used.

The following code fragment shows how the `if_fta` kernel module's `fta_error_recovery` kernel thread terminates itself by calling `thread_terminate` and `thread_halt_self`.

The `fta_error_recovery` routine is a kernel thread that starts up when the adapter becomes operational. This kernel thread resets the adapter if a fatal error occurs. The code fragment also shows the code that contains the call to `kernel_thread_w_arg`, which calls `fta_error_recovery`.

```
.
.
#include <kern/thread.h>
.
.

#define ADAP "fta"
.
.

extern task_t first_task;
.
.

struct  fta_kern_str {
.
.

short reinit_thread_started; /* reinit thread running? */
.
.

short error_recovery_flag; /* flag to wake up a process */
.
.

};
.
.

struct  ifnet {
.
.

short if_unit; /* subunit for lower-level driver */
.
.

};
.
.

fta_transition_state(struct fta_kern_str *sc,
                     short unit,
                     short state)
{
.
.
```

```
    switch(state) {
.
.
.
      case PI_OPERATIONAL: {
          int s;
          NODATA_CMD *req_buff;
          thread_t err_recov_thread;

          if (sc->reinit_thread_started == FALSE) {

              err_recov_thread = kernel_thread_w_arg(first_task,
                                         fta_error_recovery,
                                         (void *)sc);
              if (err_recov_thread == NULL) {
                  printf("%s%d: Cannot start error recovery thread.\n",
                         ADAP, ifp->if_unit);
              }
              sc->reinit_thread_started = TRUE;
          }
.
.
.
/* Perform other cases */
.
.
.
void fta_error_recovery(struct fta_kern_str *sc)
{
    struct ifnet *ifp;
    int ret_val;


    /*
     * Collect the argument left by the kernel_thread_w_arg().
     */
    ifp = &sc->is_if;

    for(;;) {
        assert_wait_mesg((vm_offset_t)&sc->error_recovery_flag,
                         TRUE,"ftaerr");
        thread_block();
        if (current_thread()->wait_result == THREAD_SHOULD_TERMINATE) {[1]
        ret_val = thread_terminate(err_recov_thread);[2]
        thread_halt_self();  [3]
        }
   }
.
.
.
/* Performs tasks to reset the adapter */
.
.
.
}
```

[1] If the `wait_result` field of the `thread` structure pointer for the current kernel thread is set to the `THREAD_SHOULD_TERMINATE` constant, you do not need to keep this error recovery kernel thread. The `fta_error_recovery` routine uses the `current_thread` routine to obtain the pointer to the currently running kernel thread.

The current_thread routine is a pointer to the currently running kernel thread. Typically, kernel modules use this routine to reference the wait_result field of the thread structure pointer tht is associated with the currently running kernel thread. A kernel module calls current_thread after calls to assert_wait_mesg and thread_block. If the kernel module needs to set a timeout, then it calls current_thread after calls to assert_wait_mesg, thread_set_timeout, and thread_block.

2 Calls the thread_terminate routine to terminate the error recovery kernel thread.

The thread_terminate routine takes a thread_to_terminate argument, which is a pointer to the thread structure for the kernel thread that you want to terminate. This pointer was returned in a previous call to the kernel_isrthread or kernel_thread_w_arg routine.

The kernel_thread_w_arg routine returns this pointer to the err_recov_thread variable. This variable is passed to thread_terminate.

Upon successfully terminating the specified kernel thread, thread_terminate returns the constant KERN_SUCCESS. If the thread structure pointer passed to the thread_to_terminate argument does not identify a valid kernel thread, thread_terminate returns the constant KERN_INVALID_ARGUMENT. On any other error, thread_terminate returns the constant KERN_FAILURE.

3 A kernel thread that terminates itself must call thread_halt_self immediately after the call to thread_terminate, because thread_terminate only prepares the self-terminating kernel thread to stop execution. The thread_halt_self routine completes the work that is needed to stop execution of the self-terminating kernel thread by performing the appropriate cleanup work.

The following code fragment shows how the if_fta module's fta_transition_state routine terminates another kernel thread (in this example, the error recovery kernel thread) by calling only thread_terminate. The fta_transition_state routine changes the state of the kernel module by performing certain fixed tasks for a given state.

```
.
.
#include <kern/thread.h>
.
.

#define ADAP "fta"
.
.

extern task_t first_task;
```

```
.
.
.
struct  fta_kern_str {
.
.
.
short reinit_thread_started; /* reinit thread running? */
.
.
.
short error_recovery_flag; /* flag to wake up a process */
.
.
.
};
.
.
.
struct  ifnet {
.
.
.
short if_unit; /* subunit for lower-level driver */
.
.
.
};
.
.
.
fta_transition_state(struct fta_kern_str *sc,
                     short unit,
                     short state)
{
.
.
.
    int ret_val;
.
.
.
    switch(state) {
.
.
.
      case PI_OPERATIONAL: {
          int s;
          NODATA_CMD *req_buff;
          thread_t err_recov_thread;

          if (sc->reinit_thread_started == FALSE) {

              err_recov_thread = kernel_thread_w_arg(first_task,
                                     fta_error_recovery,
                                     (void *)sc);
              if (err_recov_thread == NULL) {
                  printf("%s%d: Cannot start error recovery thread.\n",
                         ADAP, ifp->if_unit);
              }
              sc->reinit_thread_started = TRUE;
          }
.
.
.
/* Perform other cases */
```

```
.
.
.

/* After performing all other cases, no more need for the */
/* kernel thread */
      case PI_SHUTDOWN: {[1]

         ret_val = thread_terminate(err_recov_thread);[2]
.
.
.

void fta_error_recovery(sc)
    struct fta_kern_str *sc;
{
    struct ifnet *ifp;


    /*
     * Collect the argument left by the kernel_thread_w_arg().
     */
    ifp = &sc->is_if;

    for(;;) {
        assert_wait_mesg((vm_offset_t)&sc->error_recovery_flag,
                         TRUE,"ftaerr");
        thread_block();

/* Performs tasks to reset the adapter */
.
.
.

}
.
.
.

}
```

[1] After the `fta_error_recovery` routine completes its work and returns to `fta_transition_state`, you do not need to keep this error recovery kernel thread. The `fta_transition_state` routine sets up a case statement to handle the termination of the error recovery kernel thread.

[2] Calls the `thread_terminate` routine to terminate the error recovery kernel thread.

The `thread_terminate` routine takes a `thread_to_terminate` argument, which is a pointer to the `thread` structure associated with the kernel thread that you want to terminate. This pointer was returned in a previous call to the `kernel_isrthread` or `kernel_thread_w_arg` routine.

The `kernel_thread_w_arg` routine returns this pointer to the `err_recov_thread` variable. This variable is passed to `thread_terminate`.

Upon successfully terminating the specified kernel thread, `thread_terminate` returns the constant `KERN_SUCCESS`. If the `thread` structure pointer passed to the `thread_to_terminate` argument does not identify a valid kernel thread, `thread_terminate`

returns the constant KERN_INVALID_ARGUMENT. On any other error, thread_terminate returns the constant KERN_FAILURE.

9.7 Setting a Timer for the Current Kernel Thread

To set a time delay on the current kernel thread, call the thread_set_timeout routine.

You must call the thread_set_timeout routine as follows:

1. Lock the resource.
2. Call assert_wait_mesg to assert that the current kernel thread is about to block.
3. Unlock the resource.
4. Call thread_set_timeout to set the time of delay for the current kernel thread.
5. Call thread_block to block (put to sleep) the current kernel thread.

The following code fragment shows a call to thread_set_timeout by the if_fta module's fta_cmd_req routine. This routine puts a DMA request onto the request queue of the adapter.

```
.
.
#include <kern/thread.h> 1
.
.

struct  fta_kern_str {
.
.

struct cmd_buf *q_first; /* first in the request queue */
struct cmd_buf *q_last;  /* last in the request queue */
lock_data_t cmd_buf_q_lock; /* lock for the cmdreq queue */
.
.

}; 2
.
.

short fta_cmd_req(cmdbuf, sc, command)
struct cmd_buf *cmdbuf;
struct fta_kern_str *sc;
short command;
{
.
.

    lock_write(&sc->cmd_buf_q_lock); 3
.
.

    assert_wait_mesg((vm_offset_t)cmdbuf, TRUE, "dmareq"); 4
    lock_done(&sc->cmd_buf_q_lock); 5
```

```
    thread_set_timeout(hz * 2); 6
    thread_block(); 7
.
.
.
}
```

1 Includes the `/usr/sys/include/kern/thread.h` header file. The `thread.h` file defines structures that kernel thread routines use.

2 Defines an `fta_kern_str` data structure.

In this example, the `fta_kern_str` structure contains the following fields:

- `q_first`

  Specifies a pointer to a `cmd_buf` data structure. This field represents the first command queue in the linked list.

- `q_last`

  Specifies a pointer to a `cmd_buf` data structure. This field represents the last command queue in the linked list.

- `cmd_buf_q_lock`

  Declares a lock structure called `cmd_buf_q_lock`. The purpose of this lock is to protect the integrity of the data that is stored in the linked list of `cmd_buf` data structures. The alternate name `lock_data_t` declares the complex lock structure. Embedding the complex lock in the `fta_kern_str` structure protects the `cmd_buf` structure for any number of instances.

3 Calls the `lock_write` routine to lock the command request queue.

The `lock_write` routine takes one argument: a pointer to the complex lock structure `lock`. This lock structure is associated with the resource on which you want to assert a complex lock with write access. The `fta_cmd_req` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer.

4 Calls the `assert_wait_mesg` routine to assert that the current kernel thread is about to block.

The `assert_wait_mesg` routine takes three arguments:

- The first argument specifies the event to associate with the current kernel thread. In this call, the event that is associated with the current kernel thread is the `cmdbuf` structure pointer.
- The second argument specifies a Boolean value that indicates how the kernel thread is awakened. You can pass one of the following values:

TRUE	The current kernel thread is interruptible. This value means that a signal can awaken the current kernel thread.
FALSE	The current kernel thread is not interruptible. This value means that only the specified event can awaken the current kernel thread.

The code fragment shows that `fta_cmd_req` passes the value `TRUE`.

- The third argument specifies a mnemonic for the type of wait. The `/bin/ps` command uses this mnemonic to print out more meaningful messages about a process. The code fragment shows that `fta_cmd_req` passes the string `dmareq`.

5 Calls the `lock_done` routine to unlock the command request queue.

The `lock_done` routine takes one argument: a pointer to the complex lock structure `lock`. The `fta_cmd_req` routine passes the address of the `cmd_buf_q_lock` field of the `fta_kern_str` structure pointer.

6 Calls the `thread_set_timeout` routine to set a timer for the current kernel thread.

The `thread_set_timeout` routine takes one argument: the amount of time to wait for an event. The time is used in conjunction with the `assert_wait` routine. The `fta_cmd_req` routine passes the value `hz * 2`.

The time that you specify to wait for the event is automatically canceled when the kernel thread awakes.

The `thread_set_timeout` routine does not return a value.

7 Calls the `thread_block` routine to block (put to sleep) the current kernel thread.

10

Building a Kernel Module

This chapter discusses how to build a kernel module that you can statically link or dynamically load into the kernel:

- Build a kernel module from your source code (Section 10.1)
- Add module attributes (Section 10.2)
- Statically link a kernel module into a `/vmunix` kernel (Section 10.3)
- Dynamically load a kernel module (Section 10.4)
- Change attribute values at run time (Section 10.5)

10.1 Procedure for Building a Kernel Module

A kernel module has a file extension of `.mod`. It must contain a *module_name*`_configure` routine and a *module_name*`_attributes` table, as described in Chapter 2.

Before you can statically link or dynamically load a kernel module, you must build it. To build a kernel module, follow these steps:

1. Create a directory to contain the source files.
2. Copy the source files.
3. Create a `files` file fragment.
4. Create a `BINARY.list` file.
5. Create a `sysconfigtab` file fragment.
6. Create a `Makefile`.
7. Build the kernel module.

The following sections discuss each of these steps.

10.1.1 Step 1: Create a Directory to Contain the Source Files

Use the `mkdir` command to create a directory to contain the source files for the kernel module:

```
# mkdir /usr/sys/ExampMod
```

The directory path must start with /usr/sys. In this example, you create a directory called /usr/sys/ExampMod to contain the files for the example kernel module. You perform the work at the superuser prompt.

When you create your directory, replace ExampMod with a directory that reflects a specific name for your organization or company.

10.1.2 Step 2: Copy the Source Files

Use the cp command to copy the source and header files to the directory that you created in Section 10.1.1:

```
# cd /usr/sys/ExampMod
# cp /usr/sys/mydevelopment/example.c .
# cp /usr/sys/mydevelopment/example_util.c
# cp /usr/sys/mydevelopment/example.h .
```

The /usr/sys/mydevelopment directory is where you initially created the source files (example.c, example_util.c, and example.h).

10.1.3 Step 3: Create a files File Fragment

The files file fragment describes the source files (.c), the kernel module name, and characteristics.

Use an editor such as vi to create a files file fragment in the directory that you created in Section 10.1.1:

```
# vi /usr/sys/ExampMod/files
```

The basic syntax for the files file fragment is as follows:

```
MODULE/STATIC/module_name  standard|optional[options] Binary [1]
path/file_name.c module module_name flags[flag_list] [2]
```

[1] Declares a kernel module.

The keywords MODULE/STATIC specify that you can statically link or dynamically load the module into the kernel.

The keyword standard indicates that the module will be included in every kernel.

The keyword optional indicates that the module will be included in the kernel only if the options are present in the kernel configuration file. The keyword options can be anything such as the module name, the bus name, or the platform name.

The keyword Binary causes the sourceconfig program to generate the build rules in the BINARY Makefile. When you run this Makefile, it builds the kernel module and places it in the /usr/sys/BINARY directory.

[2] Includes a source file in the specified kernel module.

The keyword `path` is the pathname that you created in Section 10.1.1. The build process adds `/usr/sys/` to the front of the path by default. Therefore, you do not include `/usr/sys/` in your path.

The keyword `module_name` is the name of the kernel module that you are building.

The most common keyword is `flags`, followed by one or more compiler options. For more information on valid compiler options, see `cc`(1).

The following example shows the `files` file fragment for the `example` kernel module using the keyword `standard`:

```
# This is the files file fragment for the example kernel module
# using the keyword standard.
#
MODULE/STATIC/example standard Binary
ExampMod/example.c module example
ExampMod/example_util.c module example
```

This fragment instructs the build facility to compile the `example.c` and `example_util.c` source files and put them in the `example.mod` module.

The following example shows the `files` file fragment for the `example` kernel module using the keyword `optional`:

```
# This is the files file fragment for the example kernel module
# using the keyword optional.
#
MODULE/STATIC/example optional example Binary
ExampMod/example.c module example flags -g3
ExampMod/example_util.c module example flags -g3
```

This instructs the build facility to compile the `example.c` and `example_util.c` source files using the `-g3` compiler option and put the source files in the `example.mod` module. The module will be linked in the kernel only if the option `example` is in the configuration file.

10.1.4 Step 4: Create a BINARY.list File

The `BINARY.list` file contains the location of products that are not part of the standard operating system. Use an editor such as `vi` to edit or create a `BINARY.list` file in the `/usr/sys/conf` directory:

```
# cd /usr/sys/conf
# vi BINARY.list
```

The following example shows the typical contents of the `BINARY.list` file that you create:

```
/usr/sys/ExampMod:
```

The file contains the directory path that you created in Section 10.1.1. You must follow the path name with a colon (:), as shown in the example.

10.1.5 Step 5: Create the sysconfigtab File Fragment

The sysconfigtab file fragment is optional but most device drivers require one. It contains the module name and attributes. Use an editor such as vi to create a sysconfigtab file fragment in the directory that you created in Section 10.1.1:

```
# cd /usr/sys/ExampMod
# vi sysconfigtab
```

The syntax for a sysconfigtab file entry follows the stanza(4) syntax:

```
#The following illustrates a sysconfigtab entry. [1]
module_name: [2]
            Attribute1_name = Attribute1_value [3]
            Attribute2_name = Attribute2_value
            Attribute3_name = Attribute3_value
    .
    .
    .
```

[1] Includes comments at the beginning or at the end of a kernel module sysconfigtab entry. Comments are not allowed within the body of the sysconfigtab entry.

[2] Specifies the name of the kernel module followed by a colon (:). Typically, each module contains a separate sysconfigtab file entry.

[3] Specifies an attribute and its value. A valid sysconfigtab entry consists of an attribute name, an equal sign (=), and one or more values. Put each attribute name and value pair on a separate line.

In our example, the sysconfigtab file fragment for the example kernel module would be as follows:

```
# The following illustrates a sysconfigtab entry
# in the sysconfigtab file fragment for the example driver.
example:[1]
  [2] PCI_Option =  PCI_SE_Rev - 0x210, Vendor_Id - 0x1002, Device_Id - 0x4354,
                    Rev - 0, Base - 0, Sub - 0, Pif - 0 Sub_Vid - 0, Sub_Did - 0,
                    Vid_Mo_Flag - 1, Did_Mo_Flag - 1, Rev_Mo_Flag - 0, Base_Mo_Flag - 0,
                    Sub_Mo_Flag - 0, Pif_Mo_Flag - 0, Sub_Vid_Mo_Flag - 0,
                    Sub_Did_Mo_Flag - 0, Driver_Name - example, Type - C, Adpt_Config - N
    EXAMPLE_Developer_Debug = 1    [3]
```

[1] Indicates that the attributes that follow belong to the example module.

[2] Initializes the PCI adapter information attribute to allow identification of the example PCI adapters by the driver framework code. You must insert an attribute's value on one line, which may wrap to subsequent lines. However, you cannot embed new-line characters.

[3] Initializes the EXAMPLE_Developer_Debug attribute to 1, which turns on debugging messages.

You can specify any configurable attributes in /etc/sysconfigtab and declare the module's attributes in the *module_name*_attribute table in one of the module's source files.

Although not all attributes in the module's attribute table need to appear in the sysconfigtab file fragment, all attributes that appear in the sysconfigtab file fragment must be in the module's attribute table.

A module can have no attributes or it can have attributes that do not need to be modified. In those cases, the sysconfigtab file fragment is optional. If you create a file fragment, it only contains the module name followed by a colon (:).

For more information, see sysconfigtab(4).

10.1.6 Step 6: Create a Makefile

To create a Makefile, run the sourceconfig program from the /usr/sys/conf directory:

```
# cd /usr/sys/conf
# ./sourceconfig BINARY 1
```

1 Invoke the sourceconfig program followed by the BINARY configuration file name. This generates a new Makefile in the /usr/sys/BINARY directory. This Makefile contains the information to compile the standard Tru64 UNIX modules as well as the kernel module or modules that are defined in the BINARY.list file.

10.1.7 Step 7: Build the Kernel Module

Run the make program from the /usr/sys/BINARY directory to create your kernel module:

```
# cd /usr/sys/BINARY
# make example.mod 1
```

1 Invoke the make program followed by the name of your kernel module plus the .mod extension. This step creates the kernel module in the /usr/sys/BINARY directory. In this example, example.mod is the kernel module for the example module, created in the /usr/sys/BINARY directory. This step also creates a link from the /usr/sys/BINARY directory to the directory that you created in Section 10.1.1.

Invoke the make program for each module that you want to compile. The appropriate links are created as described in the previous paragraph.

You now have a kernel module that can be statically linked or dynamically loaded into the kernel.

10.2 Adding Your Module's Attributes

The sysconfigdb utility appends the sysconfigtab file fragment to the existing /etc/sysconfigtab file. Using sysconfigdb ensures that users never manually edit the etc/sysconfigtab file.

If you have a sysconfigtab file fragment, run the sysconfigdb utility. In this example, the sysconfigdb utility is invoked with the following options:

```
# sysconfigdb -a -f /usr/sys/ExampMod/sysconfigtab example
```

- The -a option

  Specifies that sysconfigdb adds the kernel module entry to the /etc/sysconfigtab database.

- The -f *filename* option

  Specifies the file name of the file fragment to be added. This option is used with the -a option.

  The kernel module name

  Specifies the name of the kernel module, example. Replace example with the name of your kernel module.

For more information, see sysconfigdb(8).

10.3 Statically Link a Kernel Module into a /vmunix Kernel

To statically link a kernel module into a /vmunix kernel, follow the steps in Section 10.1 to build your kernel module. After your kernel module is created, follow the steps outlined in this section.

10.3.1 Step 1: Create a Kernel Build Directory

To test your kernel module, create a new kernel build directory by running the doconfig program and specifying a new name for the target configuration file. This example uses the name CONRADtest.

1. Run the doconfig program from the /usr/sys/conf directory:

   ```
   # cd /usr/sys/conf
   # doconfig
   ```

2. Enter the new name for the target configuration at the following prompt:

   ```
   *** KERNEL CONFIGURATION AND BUILD PROCEDURE ***
   Enter a name for the kernel configuration file. [CONRAD] CONRADtest
   ```

   By specifing a new target configuration file name, the pre-existing configuration (CONRAD in this example) remains intact so that a working configuration is always available.

3. When you are prompted for kernel options, select any options or select none. In response to the following prompt, indicate that you do not want to edit the target configuration file:

```
Do you want to edit the configuration file? (y/n) [n]: n
```

The doconfig program will now proceed to build a new kernel in the /usr/sys/CONRADtest directory.

10.3.2 Step 2: Create a NAME.list File

To extend the basic kernel to include your new module, you need to create a *NAME*.list file where *NAME* is the name of the new kernel build directory. In the example, this file is called CONRADtest.list because the build directory name we entered in Section 10.3.1 was CONRADtest.

Use an editor such as vi to edit or create a CONRADtest.list file in the /usr/sys/conf directory:

```
# cd /usr/sys/conf
# vi CONRADtest.list
```

The CONRADtest.list file contains the pathname of the directory where the module files are located. This is the same directory which was created in Section 10.1.1. For our example, this is:

```
/usr/sys/ExampMod:
```

You must follow the pathname with a colon (:), as shown in the example. It is important that there is only one CONRADtest.list file that can exist for the CONRADtest configuration. This file must specify all additional products and modules to be linked into the kernel. In this example, we only have one module. If you have more than one module or product to test, add each pathname on its own line, followed by a colon.

10.3.3 Step 3: Run the doconfig Program

The doconfig program takes the configuration information from the /usr/sys/conf/CONRADtest file along with the information from the /usr/sys/conf/CONRADtest.list file and produces a kernel. The resulting kernel will have the new module linked into it.

Run the doconfig program from the /usr/sys/conf directory. Use the new kernel configuration name that you created in Section 10.3.1:

```
# cd /usr/sys/conf
# doconfig -c CONRADtest
```

If you specified the keyword standard in the files file fragment, take the defaults for all prompts. If you specified the keyword optional in the files file fragment, answer yes when you are prompted to edit the

configuration file. Doing so opens the configuration file. Add the following at the end of the file:

```
config_driver example
```

Replace *example* with the name of your module to include it in the linked kernel.

The doconfig program will now proceed to build a new kernel in the /usr/sys/CONRADtest directory.

10.3.4 Step 4: Copy the New Kernel to the Root Directory

Copy the new kernel from the kernel build directory to the root directory. We strongly recommend that you leave the existing and working /vmunix in the root directory and that you not overwrite it. Rather, assign a new name to the new kernel as shown in this example:

```
# cp /usr/sys/CONRADtest/vmunix /vmunix.example
```

The kernel name vmunix.example shows that the new kernel includes the example module.

10.3.5 Step 5: Shut Down and Boot the System

To test the new module, you must shut down and reboot the system with the new kernel. Make sure to specify the name of your new kernel when booting:

```
# shutdown -h now
>>> boot -fi "vmunix.example"
```

The kernel module is now part of this new kernel and can be tested with the appropriate utilities.

10.4 Dynamically Load a Kernel Module

To dynamically load a kernel module into a /vmunix kernel, follow the steps in Section 10.1 to build your kernel module. After your kernel module is created, follow the steps outlined in this section.

10.4.1 Step 1: Create the Appropriate Links

From the /var/subsys directory, create the following symbolic links:

```
# cd /var/subsys
# ln -s /usr/sys/BINARY/example.mod example.mod 1
# ln -s /subsys/device.mth example.mth 2
```

1 Create a symbolic link to the kernel module example.mod in the /usr/sys/BINARY directory. The symbolic link has the same name as the kernel module example.mod. Replace example.mod with the name of your kernel module.

2 Create a symbolic link to the kernel method file. The `device.mth` file should already exist in the `/subsys` directory. The symbolic link has the same name as the kernel module but with a `.mth` extension, `example.mth`. Replace `example` with the name of your kernel module.

10.4.2 Step 2: Load the Kernel Module

Use the `sysconfig` utility with the `-c` option to load and configure the kernel module:

```
# sysconfig -c example
```

Replace `example` with the name of your kernel module. The `-c` option dynamically loads the kernel module and configures it into the running kernel. If your module is a device driver, its device special files are also created at this time.

10.5 Changing Attribute Values at Run Time

Users, especially system administrators, may want to change attributes. You may also need to change an attribute value during kernel module development to test features of the kernel module. The `sysconfig` program allows you and your users to reconfigure a kernel module with new attribute values.

You can modify attributes at run time whether the kernel module was statically linked or dynamically loaded into the kernel.

For example, the `example` kernel module from Section 10.1.5 initializes the `EXAMPLE_Developer_Debug` attribute to 1 by default, which turns debugging messages on. To turn the messages off, call the `sysconfig` program as follows:

```
# sysconfig -r example EXAMPLE_Developer_Debug=0
```

The `-r` option directs the `example` module to be reconfigured with the new `EXAMPLE_Developer_Debug` attribute value of 0.

Not all attributes can be changed with `sysconfig`. To allow an attribute to change at run time, you must assign the `CFG_OP_RECONFIGURE` constant to the `operation` field of the attribute's `cfg_subsys_attr_t` data structure. The `sysconfig` program returns an error if you try to change an attribute that does not have this `operation` value.

The `sysconfig` program calls the module framework to change the value stored in memory. The module framework then calls the kernel module's `configure` request code to perform any other operations that are required to reconfigure the module.

Glossary

alignment
The placement of a data item in memory. For a data item to be aligned, its lowest-addressed byte must reside at an address that is a multiple of the size of the data item (in bytes).

API
Application programming interface.

application
A user-mode program that, in the context of this manual, makes various requests to the kernel modules. If a kernel module is part of a device driver, these requests typically perform I/O operations to hardware components.

argument
A variable or constant that is associated with some value that is passed to a routine. Also called a **parameter**.

atomicity
A type of serialization that refers to the indivisibility of a small number of actions, such as those that occur during the execution of a single instruction or a small number of instructions.

attribute table
An array of the `cfg_subsys_attr_t` data structure, where each instance of `cfg_subsys_attr_t` represents one table entry that defines some data item for the kernel module.

boot timeline
The series of events and dispatch points that occur as the system boots. For example, at dispatch point `CFG_PT_VM_AVAIL` virtual memory is available. See also **dispatch point**.

callback routine
The mechanism for implementing kernel modules as single binary images. Using callback routines avoids the problem of calling routines that are not yet available — statically loaded kernel modules register a callback routine that will be called later in the boot timeline. (For a static configuration, callback routines are registered to execute at dispatch points along the boot timeline.) When the routine is called, it will perform the required initialization correctly because the routines it requires will be available. See also **static mode**.

class/port driver

The class/port driver comprises two drivers. The class driver supports user interfaces while the port driver supports the hardware and handles interrupts. The driver model is always made of more than one module and it can have multiple class drivers, multiple port drivers, and some common code in a middle layer. The structure of this driver eliminates code duplication.

complex lock

A mechanism for protecting resources in an SMP environment. A complex lock achieves the same result as a simple lock but is used when there are blocking conditions. Routines that implement complex locks synchronize access to kernel data between multiple kernel threads. See also **simple lock**.

device driver

A kernel module that supports one or more hardware components. There are two driver models: the **monolithic driver** and the **class/port driver**.

dispatch point

Points along the boot timeline and post-boot that mark when certain resources or capabilities are available. Dispatch points that are initiated from user space can occur in any order. In kernel mode, these points are in strict chronological order. For example, the dispatch point that indicates that virtual memory is available (`CFG_PT_VM_AVAIL`) always occurs before locks are available (`CFG_PT_LOCKAVAIL`).

dynamic mode

The ability to add or remove software or hardware while the system is operational. For example, dynamic hardware configuration and dynamic module loading occur late in the boot timeline after these features are enabled. Contrast with **static mode**.

entry point

The address of a routine.

granularity

The size of neighboring units of memory that can be written independently and atomically by multiple CPUs. See also **atomicity**.

initialization

The tasks that incorporate a kernel module into the kernel after it has been loaded and make it available for use by the system.

interface

A collection of routine definitions and data structures that perform related functions. There are kernel interfaces and user interfaces. For example, the kernel set management (KSM) interface consists of a variety of `cfg_ksm_xxx` library routines that allow applications to manage the kernel sets. See also **routine**.

kernel module
The code and data structures in a `.mod` file, either statically linked into `/vmunix` or dynamically loaded as part of the kernel.

kernel thread
A single, sequential flow of control within a program.

load
The process of bringing a kernel module into memory and calling its `configure` routine with the `CFG_OP_CONFIGURE` request code.

lock
A means of protecting a resource from multiple CPU access in an SMP environment. See also **simple lock** and **complex lock**.

module
See **kernel module**.

module framework
The subsystem in the kernel that loads, unloads, makes other management requests, and generally keeps track of modules in the kernel.

monolithic driver
Kernel module code that is all-inclusive; supporting everything from user requests to processing interrupts from hardware.

parameter
A variable or constant that is associated with some value that is passed to a routine. Also called an **argument**.

pseudodevice driver
A driver, such as the `pty` terminal driver, structured like other drivers but that does not operate on a bus and does not control hardware. A pseudodevice driver does not register itself in the hardware topology (system configuration tree). Instead, it relies on the device driver method of the `cfgmgr` framework to create the associated device special files.

RAD
On Tru64 UNIX systems, the building blocks that make up a Non-Uniform Memory Access (NUMA) system are mapped to structures called Resource Affinity Domains (RADs). A RAD identifies the set of CPUs, memory arrays, and I/O busses that, when used together, allow the system to work most efficiently.

routine
Code that can be called to perform a function. See also **interface**.

scan
The process of looking for hardware components for the purpose of configuring hardware that is not currently configured.

simple lock
A general-purpose mechanism for protecting resources in an SMP environment. A simple lock is a spin lock. That is, routines that implement simple locks do not return until the lock has been returned. See also **complex lock**.

single binary image
A single `.mod` file that can be statically loaded as part of `/vmunix` or dynamically loaded into the kernel any time after a system boots.

SMP
See **symmetric multiprocessing**.

software synchronization
The coordination of events in such a way that only one event happens at a time.

static mode
The permanent and nonremovable parts of the kernel. Contrast with **dynamic mode**.

string
An array of characters that terminates with a null character.

subsystem
A collection of code that provides one or more interfaces or performs one or more functions.

symmetric multiprocessing
A computer environment that uses two or more CPUs. Software applications and the associated kernel modules can operate on two or more of these CPUs.

thread
See **kernel thread**.

utility
See **application**.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W